Introduction to Resampling Statistics Using *Statistics101*

Solving Statistics Problems with Simulation

John Grosberg

www.statistics101.net

Revised: 10/30/2015

Table of Contents

Preface

"The bad news is that the subject is extremely difficult. The good news is that you--and that means you--can understand it with hard thinking, even if you have no mathematical background beyond arithmetic and you think you have no mathematical capability. That's because the difficulty lies in such matters as pin-pointing the right question, but not in any difficulties of mathematical manipulation."

--Julian Simon, Resampling: The New Statistics, p.20

Here it is necessary to emphasize that the resampling method is used to solve the problems themselves rather than as a demonstration device to teach the notions found in the standard conventional approach. Simulation has been used in elementary courses in the past, but only to demonstrate the operation of the analytical mathematical ideas. That is very different than using the resampling approach to solve statistics problems themselves, as is done here.

Once we get rid of the formulas and tables, we can see that statistics is a matter of clear thinking, not fancy mathematics. Then we can get down to the business of learning how to do that clear statistical thinking, and putting it to work for you.

. . .

If you intend to go on to advanced statistical work, the older standard method can be learned alongside resampling methods. Your introduction to the conventional method may thereby be made much more meaningful.

--ibid., p 27

The first part of this book explores a number of familiar probability and statistics ideas from the viewpoint of simulation using the *Resampling Stats* computer language. The second part introduces and describes most of the features of the extended *Resampling Stats* language. I have used the process of writing the *Statistics101* program and the associated documents, including this one, to improve my understanding of statistics. If you are a student, I hope these efforts will help you learn statistics too. If you are a professional using statistics in your work, I hope that *Statistics101* with its resampling approach will be a useful addition to your kit of tools. If you find any errors or have any suggestions, I would appreciate it if you would send them to me so that I can incorporate them into the document(s). You can email me at john@statistics101.net. My website is www.statistics101.net, from which you can download the *Statistics101* program at no cost.

Introduction

Resampling Stats is the name of a statistical simulation language developed by Julian Simon and Peter Bruce. It is also the name of the computer program they developed to interpret and execute the language. The language and the program together were designed to be a simple-to-learn tool for teaching probability and statistics concepts. They were also intended to help students and professionals to get correct answers to probability and statistics problems, especially in cases where classical closed-form mathematical approaches are difficult to apply or are unavailable.

Statistics101 is a computer program that understands and executes programs written in the *Resampling Stats* language. In that sense, *Statistics101* is a “clone” of the original *Resampling Stats* program. Unlike the original *Resampling Stats* program, *Statistics101* is written in Java and is therefore able to run more or less the same on any platform where Java is supported. Moreover, *Statistics101* adds many new commands to the language.

The first part of this book, Solving Probability and Statistics Problems with Statistics101, introduces the use of *Resampling Stats* and *Statistics101* to solve probability and statistics problems. The second part, Language Basics, describes the extended *Resampling Stats* language and the use of the *Statistics101* program. Detailed information about each command in the language is accessible directly via the *Statistics101* program’s many help features. The third part, Special Techniques, describes several programming techniques that you might find useful for some common problems. The appendices include a glossary, a complete listing of all the commands and supplied subroutines with a one-line description of each, and a categorized listing of all the commands and subroutines.

How to read this book:

You may read this document in order, from start to finish. The first part, Solving Probability and Statistics Problems with Statistics101, discusses probability and statistics from the viewpoint of resampling. It illustrates the concepts using the *Resampling Stats* language, introducing the language features only as needed. As you read, you can copy the programming examples into *Statistics101* (See the tip below). Then you can run them to see the results. If you get to something in the language that you don’t understand, you can look for that topic in the later part of this document, Language Basics, or in the Glossary or in the *Statistics101* help system. The *Statistics101* help features are available from its Help menu on the program’s menu bar. The Help menu provides access to video tutorials, a searchable command and subroutine index, example programs, wizards to help with each command and a user forum.

Or, you can start with the second part, Language Basics, to get a complete picture of the extended *Resampling Stats* language and then return to the beginning of this text to learn to apply it to probability and statistics problems.

Tip: The example programs from Part 1 of this text will be found in *Statistics101* by selecting the Help>Example Programs...>Tutorial Examples menu item and choosing the appropriate file. The titles of each program example begin with the page number from the textbook. For example, the example called "p009TwoHeadsThreeTosses.txt" is from page 9.

If you are reading the text with a reader such as a Kindle, then the page numbers don't apply. The file names are descriptive of the topic in the text so that you can find the right one for where you are in the text. Open the file in *Statistics101* and follow along in your reader. A number of other typical example programs, not discussed in this text, are also distributed with the *Statistics101* program. You can access them via the Help>Example Programs menu.

For a much more comprehensive treatment of the resampling approach to probability and statistics, please see Julian Simon's free online text, *Resampling: the New Statistics*, which can be found at Peter Bruce's website, http://www.resample.com/intro-text-online/. Also, you will find a wide variety of excellent examples using *Resampling Stats* and the resampling method at http://www.Statistics101.net/PeterBruce_05-illus.pdf. All these examples will run without change in *Statistics101*.

Before reading this document, it might be helpful to watch the brief overview of the *Statistics101* program, which is accessible from the *Statistics101* menu Help>Tutorials or directly from http://www.statistics101.net/images/statistics101web_g000003.htm. The Flash® overview explains the major features of the *Statistics101* main window and menus.

NOTE: Please make sure you have the latest version of *Statistics101*. You can check whether you have the latest version by selecting the menu Help>Check for Update. You can download the most recent version from http://www.statistics101.net/statistics101web_000003.htm.

The Resampling Method

The resampling method, as described by Julian Simon, is based on the following idea:

Beneath the logic of a statistical inference there necessarily lies a physical process. The resampling methods...allow us to work directly with the underlying physical model by simulating it, rather than describing it with formulae.

You use the extended *Resampling Stats* language to describe the underlying process and then run the simulation using the *Statistics101* program to arrive at your answer.

An interesting feature of the resampling method is that you don't have to worry about fitting your problem into some pre-existing category so you will be able to know what formula to use to solve it. With resampling, you don't have to ask yourself "Is this a binomial experiment?" or "Do I need to use the t distribution, or z test or Chi-Square test...?" or "Does my problem fit the assumptions of the X formula?" That's because with resampling you don't normally need to use pre-existing formulas. You simply model the problem as it presents itself and the answers arise from the model.

As a very simple example, say you wanted to know the probability of getting exactly two heads in a toss of three coins. You could toss three coins many times, counting the number of times you got exactly two heads and dividing by the number of tosses. That is your "underlying process." That would take considerable effort and time. You could also calculate it precisely if you knew the correct formula. To know the correct formula, you would have to identify this as being in the "binomial experiment" category. Instead, with *Statistics101*, you don't care what category it belongs to. You just simulate or "model" that process as follows (text following a single quote to end of line is a comment describing what that line is doing. Comments are optional.):

```
COPY 1 2 coin                     ' let 1=head, 2=tails
REPEAT 1000                       ' repeat the following 1000 times
   SAMPLE 3 coin tosses           '   simulate toss of 3 coins
   COUNT tosses = 1 headCount     '   count number of heads (ones)
   SCORE headCount results        '   append heads count to "results"
END                               ' end of repeat
COUNT results = 2 successes       ' count how many results were exactly 2
DIVIDE successes 1000 probability ' calculate the probability
PRINT probability                 ' print the probability in output window
```

The program simulates 1000 tosses of three coins and prints out the resulting probability. You can copy the program from this document and paste it into the *Statistics101* window and then run it by clicking on the "Run" button ▶ in the *Statistics101* toolbar. The output looks like this:

```
probability: 0.368
```

Here's what the above program is doing line-by-line:

1. Put the numbers 1 and 2, representing heads and tails, into a container (called a "vector") named "coin." A vector is a list of numbers and/or names.
2. Repeat the following three commands (a, b, and c), in order, 1000 times:
   a. Take three samples at random, with replacement, from the "coin" and put these values into another container called "tosses." This is equivalent to three tosses of

a coin. On each repeat, “tosses” will contain three numbers, ones or twos, like this: (1 2 1)

   b. Count how many of the tosses were equal to 1 (i.e., heads). If “tosses” contained (1 2 1), then the “headCount” would be 2.

   c. Record the number of heads from this trial by appending it (using the SCORE command) to the “results” container, which holds the results for all trials. If headCount were 2, then 2 would be added to the end of the “results” container.

3. Count how many of the 1000 results in the results container were equal to two, i.e., two heads.
4. Calculate the probability by dividing the number of successes by the number of trials.
5. Print the probability.

Notice that if you run the program several times you will get a slightly different answer each time. That is not an error. It is the result of the fact that you are simulating a real process (tossing coins and counting the results). If you actually carried out the process several times with real coins you would also get slightly different results each time. In fact, you would be very surprised if the results were all exactly the same. The only way the results would all be the same is if the number of repetitions were infinite instead of as here, 1000. By contrast, the formula that calculates the probability of two heads out of three always gives the same result (0.375) because, in effect, it assumes an infinite number of trials. By increasing the number of trials, you get a result closer and closer to the formula’s result.

Questions for discussion:

The probabilities computed by a resampling simulation vary slightly from run to run and from the exact formula’s result. Does that invalidate the resampling method?

Assuming that your resampling simulation is correct, how can you improve its precision, i.e., how can you make the result closer to the ideal answer?

PART 1: Solving Probability and Statistics Problems with *Statistics101*

To solve probability and statistics problems using *Statistics101* you write *Resampling Stats* programs that model the process underlying your problem. You run the program "model" and it produces the desired result. For purposes of simulation, "probability" is defined as the ratio of "successful" outcomes to the number of trials. A successful outcome is simply the outcome whose probability you want to compute.

In this section, I will develop several example programs to give you the understanding needed to write your own programs. After you read this section, I recommend that you look at the excellent examples showing how to solve a wide variety of statistical problems using *Resampling Stats* and the resampling method at http://www.statistics101.net/PeterBruce_05-illus.pdf.

Probability versus Statistics

In *probability* problems, you reason from the known population to the unknown random sample. A probability problem is therefore an exercise in *deduction*, i.e., reasoning from the general (the population) to the particular (the sample). In (inferential) *statistics* problems, you reason from the known random sample to the unknown population. An inferential statistics problem is therefore an exercise in *induction*, i.e., reasoning from the particular to the general.

Deductive reasoning has the property that if the premises are sound and the reasoning process is correct, then the final deduction is correct and certain. Inductive reasoning has the property that there can be no certainty in the result because the starting premises have limited information about the general case and are yet being used to try to characterize the general case. If all the swans that you have seen have been white, would it be valid to say that "all swans are white"? So statistical results are most often stated in the form of "confidence intervals" or "hypothesis tests," which try to maximize the amount of information about the population that can be derived from a limited sample.

Building a Probability or Statistics Model

There are three main conceptual components of a probability or a statistics "model" in *Resampling Stats*: the population, the sample, and the process. *Population* refers to the entire set of objects from which a sample is drawn. *Sample* refers to a randomly selected subset of the population. *Process* refers to the method you use to select your sample from the population and how you separate desired outcomes from the undesired outcomes. I will discuss these components one at a time.

Modeling Populations

In probability problems, you know the population. In classical probability studies, the typical population may be a deck of cards, a fair coin, an urn with known numbers of different-colored balls, a pair of dice, a roulette wheel, and so on. These are useful populations to communicate probability principles with because they are widely known and very clearly defined. Given such a population, you then try to find the answer to a question about various random samples taken from that population.

In *Resampling Stats* you don't have decks of cards, you don't have colored balls and urns. All you have are numbers and names. Therefore, you have to choose sets of numbers or names to represent the relevant subsets of a population in the same proportions that they compose the population.

I use numbers instead of Named Constants in this section so you will have a clear idea of how the models work. In subsequent sections I will use Named Constants where possible, to minimize the use of arbitrary code numbers. See the section Named Constants for how to replace numbers with names to make your simulations easier to write and read.

For example, if you want to represent a fair coin, you might choose the number 1 to represent Heads and 2 to represent Tails.

You could model this in *Resampling Stats* like this:

```
COPY 1 2 coin
```

This line is a *Resampling Stats* command. A *Resampling Stats* program consists of a sequence of commands. (See Appendix 2 or Appendix 3 for a list of the commands.) The first word on a line, in this case, "COPY', is always the name of a command. Following the command name are the "arguments" that the command operates on. Arguments are the inputs and outputs of a command. This particular command copies the two numbers into the vector named coin. The two numbers are input argument vectors; the variable coin is the output argument vector. A vector is just a list of numbers. Each number in the vector is called an element. A vector may contain zero or more numbers. A vector may or may not have a name.

If you want to see the contents of coin, you can add a PRINT command to make a short program like this:

```
COPY 1 2 coin
PRINT coin
```

If you copy those two lines into *Statistics101* and run them, you will get this result:

```
coin: (1.0 2.0)
```

As you can see, the PRINT command prints the name of the vector coin and its contents to the *Statistics101* output window. A vector is represented as a list of numbers separated by spaces and enclosed in parentheses. A single number without parentheses, such as *1* or *2,* is considered to be an unnamed vector with one element.

Later, you will see how to "flip" the coin (randomly select one of the two numbers) using the SAMPLE command. But what if you want to model a weighted coin that you (somehow) know will come up Heads 6o percent of the time? You could take one of several approaches. You could use the same two numbers as above to represent Heads and Tails, but now, you have to put them in the vector in proportion to their weighting in the coin, like this:

```
COPY 6#1 4#2 unfairCoin
```

The m#n notation is a shorthand way to represent "m copies of the number n." It is easier than, but equivalent to writing:

```
COPY (1.0 1.0 1.0 1.0 1.0 1.0) (2.0 2.0 2.0 2.0) unfairCoin
```

Either way, you have put 6 ones and 4 twos into your "unfairCoin." If you then print out unfairCoin, this is the result:

```
unfairCoin: (1.0 1.0 1.0 1.0 1.0 1.0 2.0 2.0 2.0 2.0)
```

Thus if you were to select a number randomly from the `unfairCoin` vector, with each element of the vector having an equal chance, you would have a 60% chance of getting a Head (1) and a 40% chance of getting a Tail (two).

An alternative approach would be to use the numbers from 1 to 10 and say that six of them represent Heads and four of them represent Tails:

```
COPY 1,10 unfairCoin
```

In this command, the notation "n,m" is a shorthand way to represent all the integers between n and m, inclusive. So after this command executes, `unfairCoin` will contain all 10 numbers from one through 10, in order, which would print out like this:

```
unfairCoin: (1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0)
```

For the current example where our unfair coin is biased 60% heads, you would say that the numbers 1 through 6 represent Heads while 7 through 10 represent Tails. Then if you randomly select a number from the vector, you would take it as Heads or Tails depending on which group of numbers it came from. If a selected number were between 1 and 6, then you would execute the commands appropriate for heads. If the number were not between 1 and 6, then you would execute the commands for tails[1].

If your weighted coin had a less convenient imbalance, say 65% Heads, you could still use either of the above approaches, but would have to use bigger vectors to allow for the extra precision required to represent five percent increments. Here's how the two above approaches would change to model the new weighting:

```
COPY 65#1 35#2 unfairCoin
```

and

```
COPY 1,100 unfairCoin
```

The first command puts 65 ones and 35 twos into the coin. The second puts 100 numbers into it with the numbers 1 through 65 representing Heads and the others representing Tails[2].

When you model a population, you want to represent the subsets of that population that are relevant to your problem. For example, if the population is a 52-card poker deck there are several subsets that you might or might not need to model depending on the question you are trying to answer. If you want to know probabilities related to the values of the cards, but you don't care about the cards' suits, you might use

```
COPY 1,13 1,13 1,13 1,13 deck
```

[1] You could do that using Resampling Stats like this:

```
SAMPLE 1 unfairCoin result
IF result between 1 6
   ADD 1 headCount headCount   '(Or other commands to be executed for heads)
ELSE
   ADD 1 tailCount tailCount   '(Or other commands to be executed for tails)
END
```

[2]For five percent increments, you really only need 20 numbers, with 1 through 13 representing the 65% and 14 through 20 representing the 35%, but using 100 makes it easier to relate the numbers to the percents.

or as an alternative,

```
COPY 4#1,13 deck
```

This gives you 4 sets of cards from Ace (1) to King (13) but no indication of their suit.

If you care about the suit, you can use something like this:

```
COPY 1,52 deck
```

When you sample from this population, you would interpret numbers from 1 to 13 to be of one suit, say Hearts, 14 to 26 to be of another suit, etc. Or, if you cared only about the suit, you could use the numbers 1 through 4 to represent them:

```
COPY 13#1 13#2 13#3 13#4 deck
```

The examples given so far have given you some ideas that you will find useful in solving other problems, but there is no magic or "one population fits all problems" solution. Defining the right model for the population in your problem is not always easy. Careful thought is necessary, as it is in every aspect of probability and statistics. If you need to deal with populations having multiple related attributes, read the sections How to Sort Related Vectors and How to Shuffle Related Vectors.

So far we have been modeling only discrete finite populations. Discrete populations have a number of known (usually) integer values. There is just one standard discrete distribution model available as a *Resampling Stats* command, the Poisson distribution, which is modeled using the POISSON command. I won't discuss the POISSON command any further here, but it has many useful applications. If you are interested, look it up in the *Statistics101* help. The binomial distribution is another discrete distribution model, but there is no *Resampling Stats* command with which to generate binomial samples[3]. (The BINOMIALPROB command computes probabilities directly without using simulation.) Nevertheless, our coin-flipping simulations are examples that use the binomial distribution without explicitly mentioning it.

You can also model populations that consist of infinite number of measured or fractional values. These are called "Continuous Distributions." Several such models are built into the *Resampling Stats* language. The most common and familiar one would be the NORMAL command. The others are EXPONENTIAL, LOGNORMAL, PARETO, UNIFORM, and WEIBULL.

For example, if you know that the population you are dealing with is "normal" (Gaussian) and you know the population's parameters (mean and standard deviation), you can use the NORMAL command to model that population. The NORMAL command and all the other distribution commands combine the modeling of the population with the generation of samples. For example, if you wanted 10 samples from a population whose mean is 100 and standard deviation is 20, then the following command would do the job and put the 10 samples into the vector, "samples":

```
NORMAL 10 100 20 samples
```

I will discuss binomial sampling and continuous distributions in more detail later in the section Probability Distributions. But first, let's look at the second component of a probability or statistics model, *sampling*.

[3]There is a subroutine named BINOMIALSAMPLE in the *lib* directory that will generate binomial samples.

Modeling Samples

Sampling with replacement

Once you've defined your population, you will want to generate samples from it. Going back to our fair coin example, you can make one change in its population definition: since you don't care what numbers represent Heads and Tails, you can just use their names and make the code more readable. The ENUM command allows you to create names that you can use as if they were numbers. The names can even be used as elements of a vector. Now, how do you "flip" the coin? You flip the coin using the SAMPLE command. If you want one flip, you would do it this way:

```
ENUM heads tails          'Create names to enumerate the possible outcomes
COPY heads tails coin     'create model of coin (copy names into coin vector)
SAMPLE 1 coin coinSample  'randomly choose one value from coin ("flip it")
PRINT coinSample
```

This SAMPLE command chooses one value at random from the two in `coin` and puts a copy of that choice in `coinSample`. That is equivalent to flipping one coin. If you run the above three-line program several times, you will see that sometimes `coinSample` is a `heads`, other times it is a `tails`.

If you change the command to:

```
SAMPLE 3 coin coinSample
```

then you have sampled `coin` three times, with replacement, and copied the three results into the `coinSample` vector. This is equivalent to flipping one coin three times or flipping three coins one time. If you run the program with this new SAMPLE command you will get output that looks like this:

```
coinSample: (tails tails heads)
```

Notice that even though `coin` has only two values in it, you can sample it as many times as you want, because SAMPLE does not remove its chosen value from the vector. This method of sampling, where choosing a particular sample one time doesn't keep you from choosing that same sample again, is called "sampling with replacement." If, after you draw a card from a deck, you replace it back into the deck before you make the next draw, that is another example of sampling with replacement.

In general, then, when you want to sample with replacement from a finite population, the SAMPLE command is the one to use.

Sampling without replacement

In the previous section you saw how to use the SAMPLE command to simulate random sampling with replacement from a given population. In this section you will see how to simulate random sampling without replacement. "Sampling without replacement" means that once you choose a particular item from a population, you do not replace it into the population. Thus, you can't choose that same item in subsequent choices.

Let's go back to our card deck. Suppose you wanted to simulate dealing a 5-card hand to discover the probability of getting two aces in the hand. First, you would decide how to simulate the deck (the population). In this case, you don't care about the cards' suits, only their values. So you decide to use this model:

```
COPY 1,13 1,13 1,13 1,13 deck       '1 = Ace
```

Now you want to simulate dealing the hand of five cards, but first you notice that the numbers in the deck are all in the very predictable order depicted in the COPY command. They are not shuffled. So you look at the list of *Resampling Stats* commands in Appendix 2 or Appendix 3 and find a command named SHUFFLE. After reading the help text for it, you decide it is a good fit. So you add that command to your incipient program to get:

```
COPY 1,13 1,13 1,13 1,13 deck       '1 = Ace
SHUFFLE deck deck                   'shuffle deck, put result in deck
```

If you put print statements after each line you will see the results of the shuffle.

Before shuffle:

```
deck: (1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 ... )
```

After shuffle:

```
deck: (7.0 12.0 12.0 6.0 10.0 2.0 7.0 13.0 4.0 9.0 ...)
```

Each time you shuffle `deck` you will get a different order of its contents. Now the simplest way to “deal” the cards is to take the first five cards from the deck. The TAKE command is designed to copy desired elements from its input vector to its output vector. The desired elements are specified by another vector. So the program becomes:

```
COPY 1,13 1,13 1,13 1,13 deck  '1 = Ace
SHUFFLE deck deck              'shuffle deck
TAKE deck 1,5 hand             'Take first 5 from deck; put in hand
```

This TAKE command can be read as “From `deck`, take elements 1 through 5 and copy them to `hand`.” So now you have dealt a hand. What you have accomplished up to this point is to take a sample without replacement from a population. For any case, not just a deck of cards, where you have to sample a population without replacement, you can use the SHUFFLE and TAKE commands in tandem.

You might be objecting, “That’s not the way you deal cards. Usually, you deal one card to each player before giving the next card to the first player. You don’t give the first five cards to player one, then the next five to player two, and so on.” That is true. However, if the deck is truly shuffled, the probabilities resulting from the two ways of dealing are identical. Although the two ways of dealing will make a difference to a particular player in a particular game, on average the results will be the same. And on average, if many games are played, the method of dealing (as long as it is honest) will not affect the outcome for any player. If that isn’t intuitively clear to you after thinking about it for a while, try writing a *Resampling Stats* program that compares the two methods. Hint: you’d be comparing the results of using this command

```
TAKE deck 1,5 hand   'Take first five “cards” from deck
```

with using this one:

```
TAKE deck (1 5 9 13 17) hand  'skip cards that go to 3 other players.
```

In general, to sample without replacement, you will usually use the SHUFFLE and TAKE commands together.

Modeling Processes

"Process," the third part of a model, refers to the method you use to select your sample from the population and to separate the desired outcomes from the undesired outcomes. In our current problem, the process is to record how many times the hand contains two aces. This means you have to shuffle the deck, draw a hand, count how many aces are in the hand, and if there are two, record that as a success. You do this repeatedly, and then calculate the proportion of successes out of the number of trials as the desired probability.

Having come this far, you might as well complete the program. How many aces does the hand contain? You need to count how many ones are in the simulated hand. You find the COUNT command in the list of commands in Appendix 2 or Appendix 3 and see that it is suitable. It counts how many elements of a vector pass a certain test. Here you want it to count how many aces are in a hand:

```
CONST 1 ace                     'Assigns the name "ace" to the number 1
COPY 1,13 1,13 1,13 1,13 deck  'create the deck
SHUFFLE deck deck               'shuffle deck
TAKE deck 1,5 hand              'Take first 5 from deck; put in hand
COUNT hand = ace aceCount       'Count the aces (ones) in the hand
```

I've added a named constant, "ace," using the CONST command so I can use it in the COUNT test instead of using a bare "1" there. If COUNT finds two aces, you want to record that as a success. That requires using two new commands, IF and LET. Adding those to the program gives this:

```
CONST 1 ace                     '1 = Ace
COPY 0 successCount             'initialize a variable to count successes
COPY 1,13 1,13 1,13 1,13 deck  'create the deck
SHUFFLE deck deck               'shuffle deck
TAKE deck 1,5 hand              'Take first 5 from deck; put in hand
COUNT hand = ace aceCount       'Count the aces (ones) in the hand
IF aceCount = 2                 'If the hand had 2 aces...
   LET successCount = successCount + 1 '...increment the successCount vector
END
```

The LET command expresses mathematical formulas using familiar operator symbols such as +, -, *, / and some command names as will be described later in the section on The LET Command. In this example, you can read the LET command as "Let `successCount` equal its current value plus one." Under the control of the IF command, the LET command increments the `successCount` vector only if `hand` contained two aces.

The IF command can control many lines, so it must be told where its control ends. That's what the END command is for: it marks the end of the IF command's scope. So far, this is one trial consisting of one shuffle and one hand. That's not enough to determine a probability, so you decide you need to repeat the process many times. The REPEAT command is designed for this purpose. You end up with this:

```
CONST 1 ace                        '1 = Ace
COPY 0 successCount                'initialize a variable to count successes
COPY 1,13 1,13 1,13 1,13 deck     'create the deck
COPY 1000 numberOfTrials           'could use CONST here instead of COPY
REPEAT numberOfTrials              'Repeat the following 1000 times
   SHUFFLE deck deck               'shuffle deck
   TAKE deck 1,5 hand              'Take first 5 from deck; put in hand
   COUNT hand = ace aceCount       'Count the aces (ones) in the hand
   IF aceCount = 2                 'If the hand had 2 aces...
      LET successCount = successCount + 1 'increment the successCount vector
   END
END
LET probability = successCount / numberOfTrials
PRINT probability
```

The REPEAT command's scope, like that of the IF command, terminates with an END command. The REPEAT command is one of four "looping" commands. The other three are FOREACH, WHILE, and UNTIL. Each of the looping commands has a different way of deciding how many times to repeat the commands in its scope.

Notice that this program uses a variable name, `numberOfTrials`, for the number of trials. The number of trials is used in two places in the program and if you wanted to change it, you would have to remember to change it in both places. Using a variable means that you only have to make the change in one place, the line where the variable is given its initial value. The program also uses the CONST command to create a Named Constant.

One run of this program produced the answer:

```
probability: 0.039
```

Probability Examples

Now that we've collected some ideas on how to model populations, samples, and processes in *Resampling Stats*, we can proceed to apply these ideas to problems in probability.

Independent Events

Independent events are those for which the outcome of one trial is not related or dependent on the outcome of another. Independent events often can be modeled by sampling with replacement. As the simplest example, take the fair coin. A coin doesn't have any memory of its past so each toss is "independent" of any and all previous tosses. By definition, the fair coin will come up heads on average 50% of the time. Say you wanted to know the probability that if you flip a coin 3 times you will get exactly two heads. You could answer that question by flipping the coin thousands of times, recording how many times each sequence of three flips had exactly two heads and then dividing that number by the number of groups of three you had tossed—if you had enough time and interest. Or you could apply the classical solution if you know that. The classical solution is achieved by counting the number of ways that three coins can come up with two heads (3) and dividing by the number of different ways that three coins can fall (8) to get the probability of 3/8, or 0.375. With *Statistics101* you use the method of simulating the thousands of coin tosses and recording the results. You have seen a program that performs this simulation before in this document. I repeat it here for ease of reference.

```
ENUM heads tails
COPY heads tails coin
COPY 1000 numberOfTrials
REPEAT numberOfTrials
   SAMPLE 3 coin trial      ' flip coin 3 times
   COUNT trial = heads headcount
   SCORE headcount headCounts
END
```

```
COUNT headCounts = 2 successCount
LET probability = successCount / numberOfTrials
PRINT probability
```

Running this program three times produced the following outputs:

```
probability: 0.367
probability: 0.375
probability: 0.371
```

You know from the classical method that the correct answer is 0.375. Our simulation got results that hover around that value. If you needed more precision, you could do more than 1000 trials. But an important result of using simulation is that you clearly see the variation in the results. While 0.375 is the "exact" answer, in the real world you will rarely experience exactly 0.375. It is very important in probability and statistics to keep that variability of results in mind. After all, probability is probability, not certainty.

For variety, let's look at another example of independent events. This example was found in *CliffsQuickReview Statistics*. A dartboard is divided into twenty equal wedges, ignoring the bull's eye. Only six of the twenty wedges are worth points, so the probability of scoring on one throw is 6/20, or 0.3, assuming your throws always hit the board. What is the chance of hitting at least one scoring region in three consecutive throws? (From *CliffsQuickReview Statistics*, p.48).

```
ENUM score noScore
COPY 3#score 7#noScore scoringProbability
COPY 1000 numberOfTrials
COPY 0 successCount
REPEAT numberOfTrials
   SAMPLE 3 scoringProbability throws
   COUNT throws = score numberOfScores
   IF numberOfScores >= 1
      LET successCount = 1 + successCount
   END
END
LET probability = successCount / numberOfTrials
PRINT probability
```

The first task is to create a simulation for the probability of scoring. That was done by putting three "`score`" tokens and seven "`noScore`" tokens into a `scoringProbability` vector. This is our "population." It is as if you had labeled three pieces of paper with the word "score" and seven with "noScore" and put them in a hat. Next, you want to simulate the process. Using the SAMPLE command to draw three tokens at random, with replacement, from the `scoringProbability` vector simulates three independent throws at the dartboard. The three throws constitute one trial. Next, you want to see if this trial meets the criterion for success, i.e., did it have at least one throw that scored? So you count the number of times `score` occurs in your sample of three. If that number is at least one (>=1), then you record it as a success by incrementing the `successCount`. After doing this 1000 times, you divide that sum by the number of trials to obtain the probability.

Dependent Events

In the previous problem of the two heads in three coin tosses, all the events were independent of one another. In other words, if one of the three tosses came up heads, that had no influence on the probability of any of the other tosses. Here's a problem where the outcome of one event affects the probability of the second. What is the probability of drawing an ace at random from a deck of cards, and then on your second draw drawing another ace? Clearly when you start, there are 4 aces in the deck of 52 cards. So the probability of getting an ace is 4/52. Once you've drawn an ace, there are only 3 aces left in the deck of now 51 cards. So on the second draw, the chance of getting an ace is 3/51. This is an example of sampling without replacement. The selection process results in the probability of the second draw depending on the outcome of the first draw. Events related in this way are called "dependent events." Dependent events often can be modeled by sampling without replacement. As the simplest example, take the fair coin. You might know that the rule that applies here is to multiply the two probabilities together. Thus the answer to the question would be 4/52 x 3/51 = .00452. Here is a *Resampling Stats* program that simulates drawing 2 cards from the deck many times:

```
ENUM 1 ace
COPY 1,13 1,13 1,13 1,13 deck
COPY 10000 numberOfTrials
COPY 0 successCount
REPEAT numberOfTrials
   SHUFFLE deck deck
   TAKE deck 1 firstCard
   IF firstCard = ace
      TAKE deck 2 secondCard
      IF secondCard = ace
         LET successCount = 1 + successCount
      END
   END
END
LET probability = successCount / numberOfTrials
PRINT probability
```

This program first creates a simulated card deck by copying four sets of sequences from 1 to 13 into a vector called "deck." These 4 sets each represent one of the suits in the deck. I let the number 1 represent the Ace. Thus, there are 4 aces in the deck of 52 cards.

Next it creates a variable, `numberOfTrials`, containing the number of repetitions to perform. It uses this variable as the argument of the REPEAT command. Notice that it also uses this variable later in the program when it divides to calculate the probability. This way, if you want to change the number of repetitions, you only need to change it in one place. Then it initializes a counter, `successCount`, that will count the number of successes.

Next the program uses the SHUFFLE and TAKE commands. SHUFFLE randomly rearranges the elements in the `deck` vector. TAKE copies the first value into the variable `firstCard`. The IF command compares `firstCard` to `ace` (1) and if it is an Ace, TAKE looks at the second card. If this second card is also an Ace, the LET command adds one to the counter. The SHUFFLE and TAKE commands used together in this way accomplish the goal of sampling without replacement. When the REPEAT command completes its many iterations, the `successCount` will contain the number of times that two aces were drawn. The final LET command divides the number of successes by the number of trials to yield the `probability`. Finally, PRINT outputs the `probability`.

Running this program a few times produced these results:

```
probability: 0.0052
probability: 0.0053
```

```
probability: 0.0043
probability: 0.0045
```

This is a low probability occurrence so it is necessary to use a large number of trials to get a reasonably accurate result.

Summarizing, the general resampling approach in probability problems is to simulate the universe, take repeated samples from that universe, test the samples for “success,” then find the ratio of the number of successes to the total number of trials. That ratio is the desired probability value. For independent events, use the SAMPLE command, which simulates sampling with replacement. For dependent events, use the SHUFFLE command with the TAKE command, which together simulate sampling without replacement.

You will find many probability examples in the folders named “GeneralExamples” and “SubroutineExamples” that are in the “ResamplingPrograms/ExamplePrograms” folder in your *Statistics101* installation folder.

Question for discussion:

Is there any probability problem that cannot be solved by simulation that can be solved by other, more traditional means?

Probability Distributions

In addition to populations such as decks of cards, dice, coins, etc., there are a number of more abstract populations that are often useful. These populations are described mathematically by probability distributions.

A probability distribution describes the values that a random variable can take on and their associated probabilities. There are two broad classes of probability distributions: discrete and continuous. A *discrete distribution* is the distribution of a discrete random variable. A discrete random variable has a finite number of possible values, each with an associated probability. A *continuous distribution* is the distribution of a continuous random variable. A continuous random variable has an infinite number of possible values. Since it has an infinite number of values, the probability of any one value is effectively zero. Having zero probability for every value is not particularly useful, therefore the probability distribution for a continuous random variable is defined so that it gives the probability that a value will fall within some interval. There is one constant rule that applies to all probability distributions, discrete and continuous: the sum of the probabilities for all the possible values must be unity.

We will look at the two types of distributions in turn.

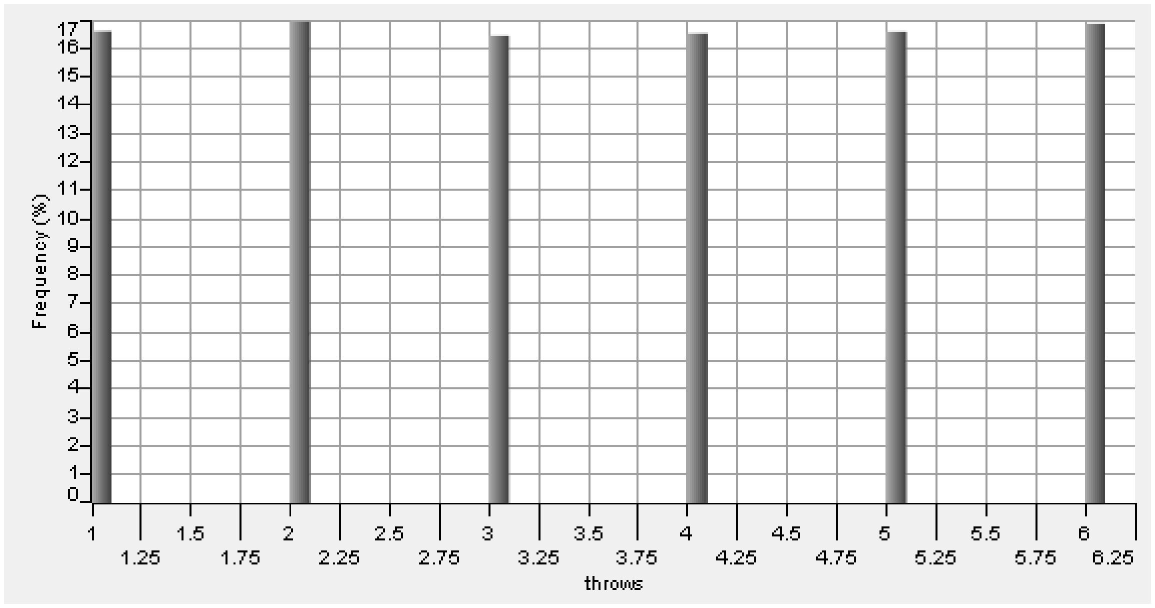
Discrete Distributions

Discrete Uniform Distribution

An example of a discrete distribution would be the result of tossing a die. Each face of the die has a 1/6 probability of occurring. So the probability distribution for the die consists of the 6 pairs of numbers, with the first number being the face and the second the probability: (1, 1/6), (2, 1/6), (3, 1/6), (4, 1/6), (5, 1/6), (6, 1/6). Since the probabilities of all the possibilities are equal, this is called a “uniform” distribution, or more specifically, a “discrete uniform” distribution. Here is a program that will generate a histogram for rolling a fair die:

```
COPY 1,6 die
SAMPLE 100000 die throws
HISTOGRAM binsize 0.1 percent throws
```

This HISTOGRAM command is using two optional keywords. The first, **binsize** n, where `n` is any positive number, sets the width of a histogram bin to `n`. The second keyword, **percent**, causes the frequencies to be displayed in percents rather than in raw numbers. For more of HISTOGRAM's options, refer to the program's help file. Here is the resulting histogram. You can see that all the bins are very close to the same height (1/6 or about 16.7%), as expected for a uniform histogram.



Other discrete distributions include the binomial distribution and the Poisson distribution. We won't discuss the Poisson distribution here (*Statistics101* has a POISSON command that you can learn about in the program's help documentation), but the next section will look at the Binomial Distribution.

Binomial Distribution

A Binomial distribution is the result of a process which has only two possible outcomes (for example, heads and tails, win or lose, score a touchdown or not), where each outcome has a complementary probability (i.e., they must sum to one; for the coin example, 0.5 for heads and 0.5 for tails), and the outcome of each trial is independent of the outcomes of any previous trials.

We've seen examples with fair and unfair coins that simulate the conditions for a binomial distribution. As you will recall, it wasn't even necessary to identify those as examples of a binomial distribution. The results just followed automatically from the simulation.

In general, you simulate a binomial population using a vector that contains multiples of two different values in quantities proportional to their probabilities. For example, if the probability of success is 0.3, you might simulate the population like this:

```
COPY 3#1 7#2 binomialPopulation  '1 = success, 2 = failure
```

This above command results in `binomialPopulation` having these contents:

```
binomialPopulation: (1.0 1.0 1.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0)
```

Then, you use the SAMPLE command to select a given sample size with replacement from the binomial population:

```
SAMPLE 15 binomialPopulation mySample
```

Then, you count the number of successes and save it in a scoring vector:

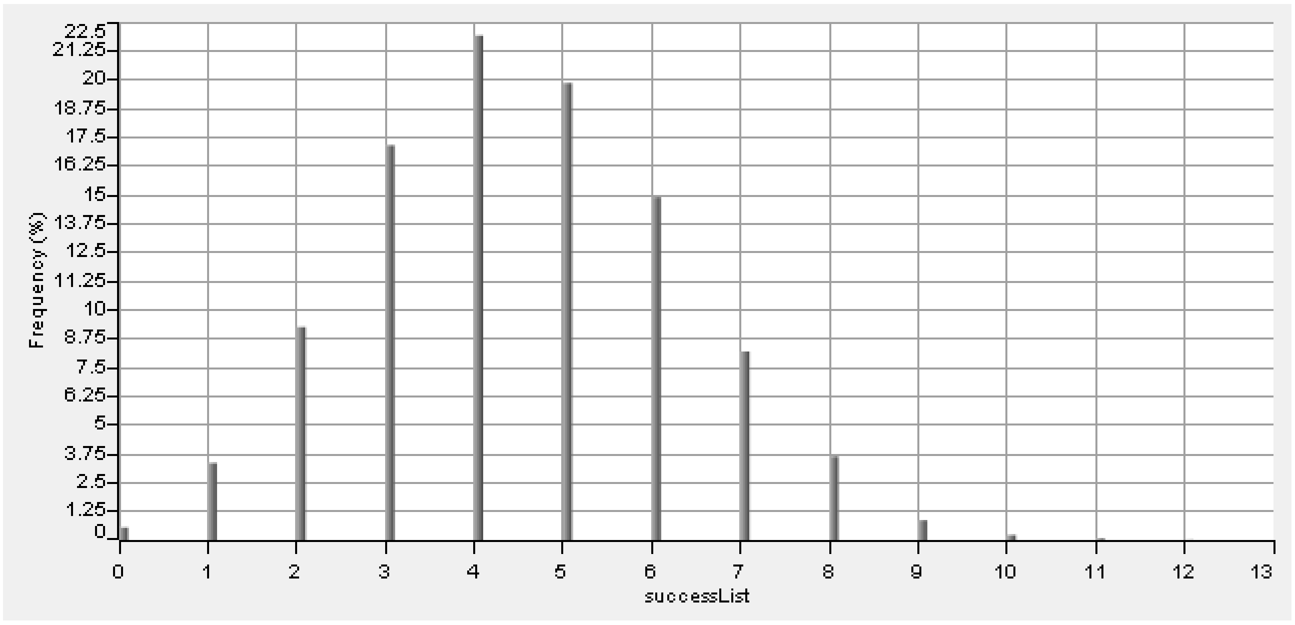
```
COUNT mySample = 1 successCount  'Count how many 1s are in mySample
SCORE successCount successList   'Append successCount to successList
```

Repeat the above three commands say 10000 times and successList can be put in a histogram to show the binomial distribution results.

Putting it all together, you have the following program:

```
COPY 3#1 7#2 binomialPopulation  '1 = success, 2 = failure
REPEAT 10000
   SAMPLE 15 binomialPopulation mySample
   COUNT mySample = 1 successCount  'Count successes, how many 1s in mySample
   SCORE successCount successList   'Append successCount to successList
END
HISTOGRAM percent binsize 0.1 successList
```

which produces the following histogram:



Statistics101 doesn't have a built-in command that generates binomial samples. (The BINOMIALPROB command computes probabilities, it doesn't produce samples.) And most of the time you will generate binomial samples using several commands as shown above. But you can also use another approach that was discussed earlier, that is to use a sequence of numbers, say the integers from 1 to 100, and use an IF command or a test to set the division point between success and failure. For example, if you wanted to generate binomial samples where the success probability is 0.6 or 60%, you can do something like this:

```
'Program to select a given number of samples from
'a binomial distribution with a given percent probability
```

```
'of success.
ENUM success failure

COPY 60 successPercentProb  'Arbitrarily choose 60%
COPY 10 sampleSize
REPEAT sampleSize
   SAMPLE 1 1,100 result
   IF result <= successPercentProb
      SCORE success sample
   ELSE
      SCORE failure sample
   END
END
PRINT sample
```

The ELSE command divides the IF command into two parts, the first part is executed only if the IF test evaluates to true. The second part, after the ELSE, is executed only if the IF test evaluates to false. This program produces the following output (yours will differ due to the randomness of the SAMPLE command):

```
sample: (success failure success failure success failure success failure
success success)
```

If you use the above program to generate binomial samples for success probabilities less than 1% or for non-integer percent probabilities such as 25.5%, it will not generate accurate answers. You would have to have more than just 100 numbers in the population to accommodate those cases. For example, to be accurate to within 0.5 percent you would need to use 200 numbers. Nevertheless, as you will see shortly, this method can be generalized to work for any level of precision by using an infinite population instead of a finite sequence of integers.

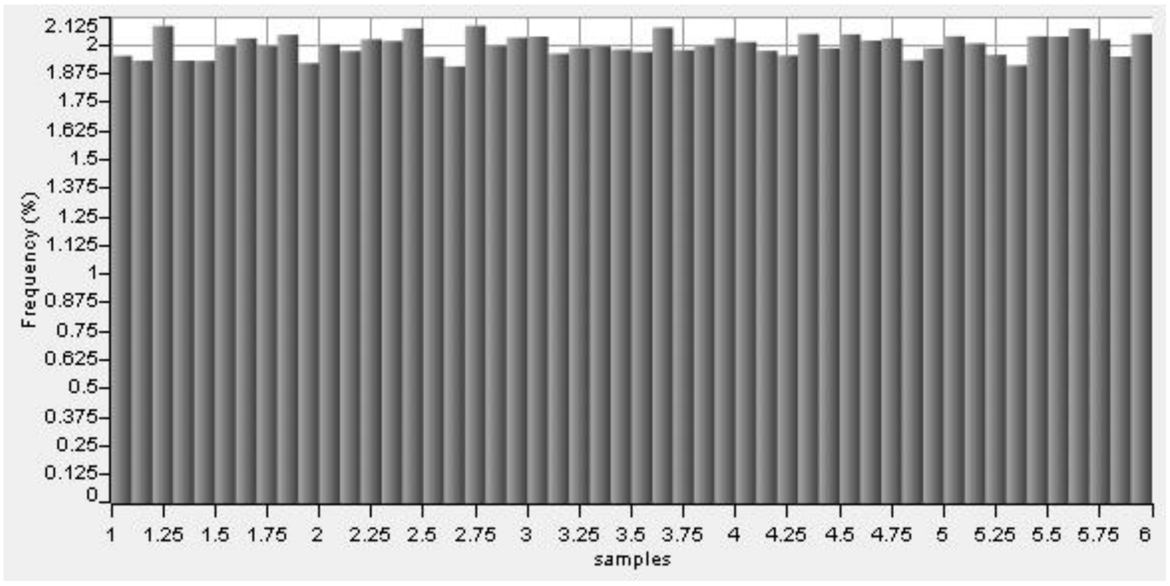
Continuous Distributions

Continuous Uniform Distribution

As stated earlier, a continuous distribution has an infinite number of possible values. One simple example of a continuous distribution is the "continuous uniform" distribution. Like the discrete uniform distribution, all its possible values have equal probabilities, but there are an infinite number of possible values so the probability of any one specific number will be effectively zero. You can imagine a die with an infinite number of sides. A continuous uniform random variable is simulated by the UNIFORM command in *Resampling Stats*. Here is a program to generate a histogram of a uniform continuous random variable between 1 and 6.

```
UNIFORM 100000 1 6 samples
HISTOGRAM percent binsize 0.1 samples
```

And here is the resulting histogram.



Probability Density

You can see, in contrast to the earlier discreet uniform distribution based on rolling a die, that there are no gaps between the bins. That's because there is an infinite number of numbers between 1 and 6, not just the six integers, and each has some chance of being selected.

The percent frequency for any one of the bins is computed as:

percent frequency = (number of samples in the bin) / (total number of samples) * 100

which is, by definition, the same as:

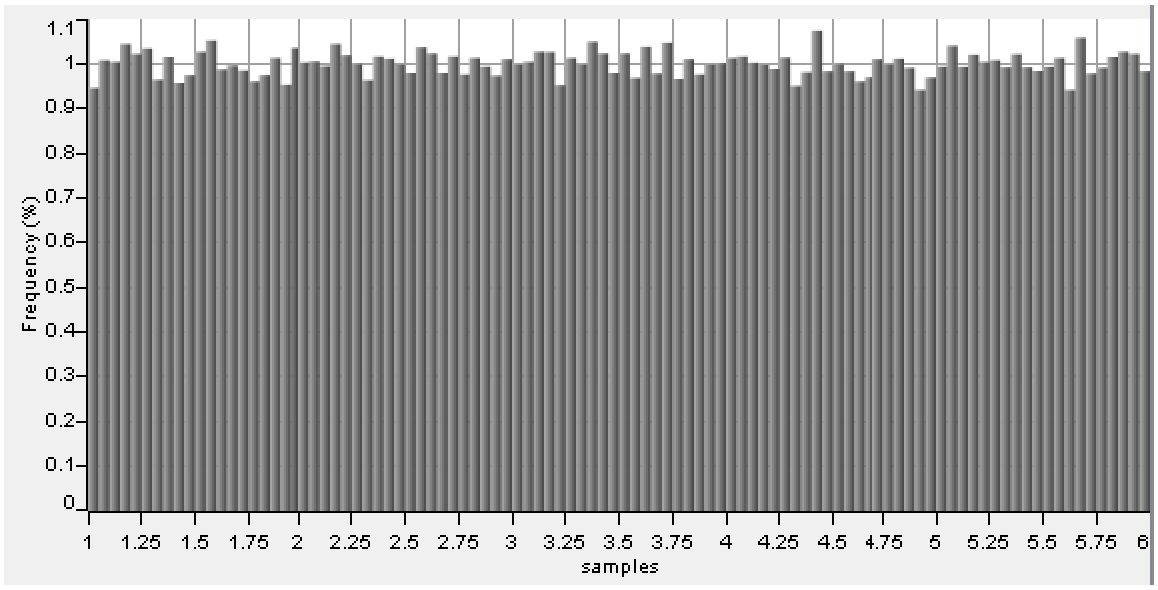
percent frequency = (probability of being in this bin) * 100.

Therefore, using percent on the Y scale allows you to read the Y scale as if it were a probability expressed as a percentage. For example, the probability that a randomly chosen number between 1 and 6 will fall in the first bin (or any bin) is about 2%. If you wanted to know the probability that a number chosen randomly would be in the first two bins, you would just add the probabilities of the two bins, which means to add their bin heights. This would give you a probability of about 4%. More generally, if you wanted to calculate the probability for any interval, say between 3 and 4, you would just add the heights of all the bins in that interval. Since for a uniform random variable they are all the same height, you can just multiply the number of bins, 10, by the height, 2%, to get 20% as the probability over the interval from 3 to 4.

In this example there are fifty bins. Since the bins are all the same width and this is a uniformly distributed variable, they each get about the same share of the samples. So the probability that a sample will fall in a particular bin is one fiftieth (0.02 or 2%) of the probability (1.0) that it falls within the range one to six. If you cut the bin size in half, doubling the number of bins, with these commands,

```
UNIFORM 100000 1 6 samples
HISTOGRAM percent binsize 0.05 samples
```

then the height of each bin will also be cut in half because now the probability of falling into any one of the smaller bins is one half of what it was with the larger bins. Here is the histogram for this case:



As you can see, the height is now about 1.0 percent. So to calculate the probability that a randomly selected number would fall between 3 and 4, now you must multiply 1% by 20 since there are 20 bins between 3 and 4. Once again, you get 20%.

Now, if you were to continue to shrink the bin width so that it approached zero, the bin heights would also approach zero because each bin would hold a smaller percentage of the population. Note that this does not happen for discrete distributions, such as the six-sided die or the binomial. Therefore, for continuous distributions it is more common divide the height by the bin width to produce what is called a probability "density" curve. Here is how that would look in Resampling Stats:

```
UNIFORM 100000 1 6 samples
HISTOGRAM density binsize 0.05 samples
```

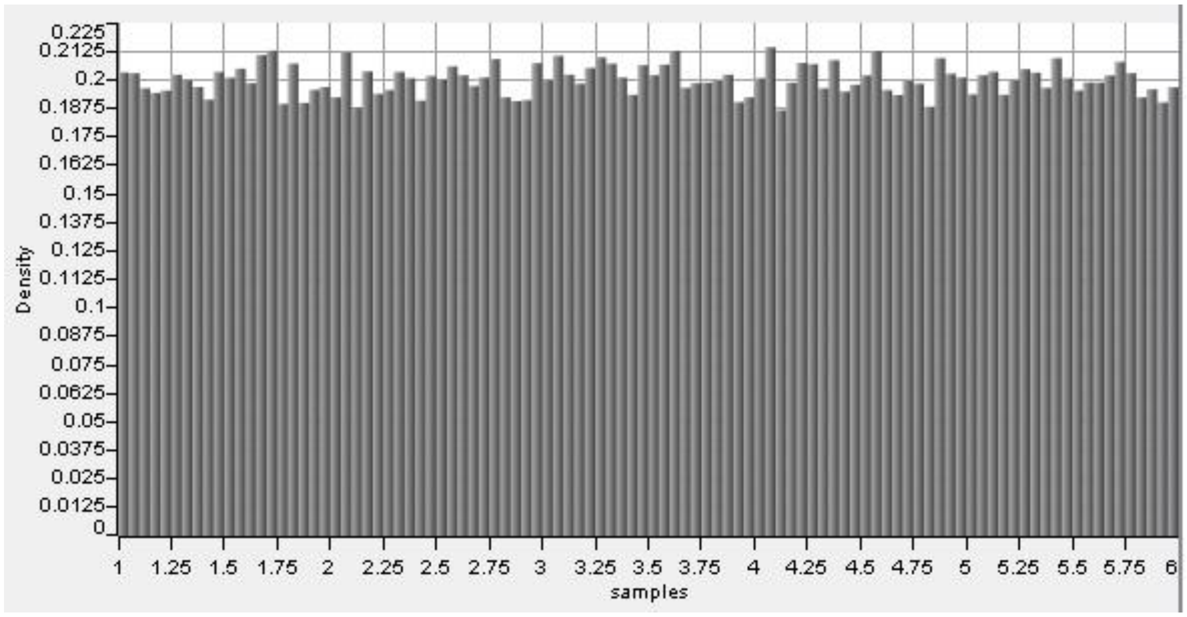
Using the **density** keyword, the histogram divides each bin's count by the bin's width, like this:

density = (number of samples in the bin) / (total number of samples) / (bin width)

or,

density = (probability of being in this bin) / (bin width)

The graphical result of the above two-line program is shown next.



You can change the bin size in the program and notice that the bin heights will remain hovering around 0.2 no matter what size you choose. But remember that the Y axis is no longer a probability. To get the probability, you have to multiply the bin height, or density, by the bin width. This is the area of the bin. To find the probability that a sample would fall between any two points on the X axis, you would sum the areas of the bins between those two points to get the total area between the points. Conceptually, as the bin width approaches zero, the summation of the bins between the two points approaches the integral of the density curve between the two points.

Why is the density for this example 0.2? Because the area of the entire density curve, being a probability, must be 1.0. And since the width is 6 - 1 = 5, then for the simple case of this uniform continuous distribution, the height must be 1.0/5 = 0.2.

The various probability distributions are often useful in Monte Carlo simulations in fields outside of probability and statistics. But when solving probability and statistics problems using the resampling method, there is seldom a need to use the predefined probability distributions unless you happen to know that one of them characterizes your population of interest.

A Simple Application of the Continuous Uniform Distribution

The continuous uniform distribution can be used to generate samples for other types of distributions, even discrete ones. In our earlier discussion of the discrete binomial distribution, I developed a short program that generated samples from such a distribution. The program worked by choosing samples from the sequence of integers between 1 and 100. That method had the limitation that it would only work for integer values of percent probabilities. But now that we know about the UNIFORM command, we can use it as an infinite number of integer and non-integer values to overcome that limitation. Now, instead of choosing from 100 integers, we will be choosing any number from the interval between 0 and 100. Here is the program modified to use the UNIFORM command.

```
'Program to select a given number of samples from
'a binomial distribution with a given percent probability
```

```
'of success.
ENUM success failure

COPY 60 successPercentProb
COPY 10 sampleSize
REPEAT sampleSize
   UNIFORM 1 0 100 result
   IF result <= successPercentProb
      SCORE success sample
   ELSE
      SCORE failure sample
   END
END
PRINT sample
```

This program uses the continuous uniform distribution to generate a number between 0 and 100. The number will most likely not be an integer. If the number is less than or equal to the `successPercentProb`, then that is scored as a `success`; otherwise it is scored as a `failure`. It repeats this selection as many times as needed to get the required `number` of samples. Each sample is appended by SCORE to the `sample` vector. When the program completes, `sample` will contain the random selection of the enums, `success` and `failure`.

Introducing Subroutines

Since the need for computations like these might occur repeatedly, it would be convenient if there were a way to package them as a command so that you only need one line to do the whole job. The way to package a set of commands is to put them in a subroutine. A subroutine is like a short program that can be invoked by another program. A subroutine is declared using the NEWCMD command, which gives the subroutine a name and a list of argument names. You invoke a subroutine by using its name as a command in exactly the same way you invoke built-in commands.

So I'll revise the above program to make it a subroutine that generates samples for the general binomial case and while I'm at it, I will make it use actual probability numbers (i.e., between 0 and 1) instead of percent probabilities:

```
'Subroutine to select a given number of samples from
'a binomial distribution with a given probability
'of success.
ENUM success failure
GLOBAL success failure
NEWCMD BINOMIALSAMPLE sampleSize successProb sample
   CLEAR sample
   REPEAT sampleSize
      UNIFORM 1 0 1 result
      IF result <= successProb
         SCORE success sample
      ELSE
         SCORE failure sample
      END
   END
END
```

The NEWCMD command introduces the subroutine. Its first argument, BINOMIALSAMPLE, is the name to be given to the subroutine. Following the subroutine name are three arguments that are analogous to command arguments. They are called “dummy arguments” because when the subroutine is later invoked, these arguments will be replaced with your actual arguments that may have different names. All the commands between the NEWCMD and the final END command constitute the subroutine.

The GLOBAL command makes the constants `success` and `failure` visible within the subroutine. It is needed because subroutines can’t otherwise see variables that are defined outside of them.

The CLEAR command is necessary to avoid appending the results of the current invocation of BINOMIALSAMPLE to leftover results of previous invocations if the subroutine were to be invoked more than once in the same program.

You’ll notice that I changed the limits on the UNIFORM command to be 0 and 1 instead of 0 and 100. That change makes it so you can use actual probability numbers instead of percentages for the success probability.

Now you can create samples from a binomial distribution with one command:[4]

```
BINOMIALSAMPLE 10 0.5 sample
```

This command produces a result that looks like this when `sample` is printed:

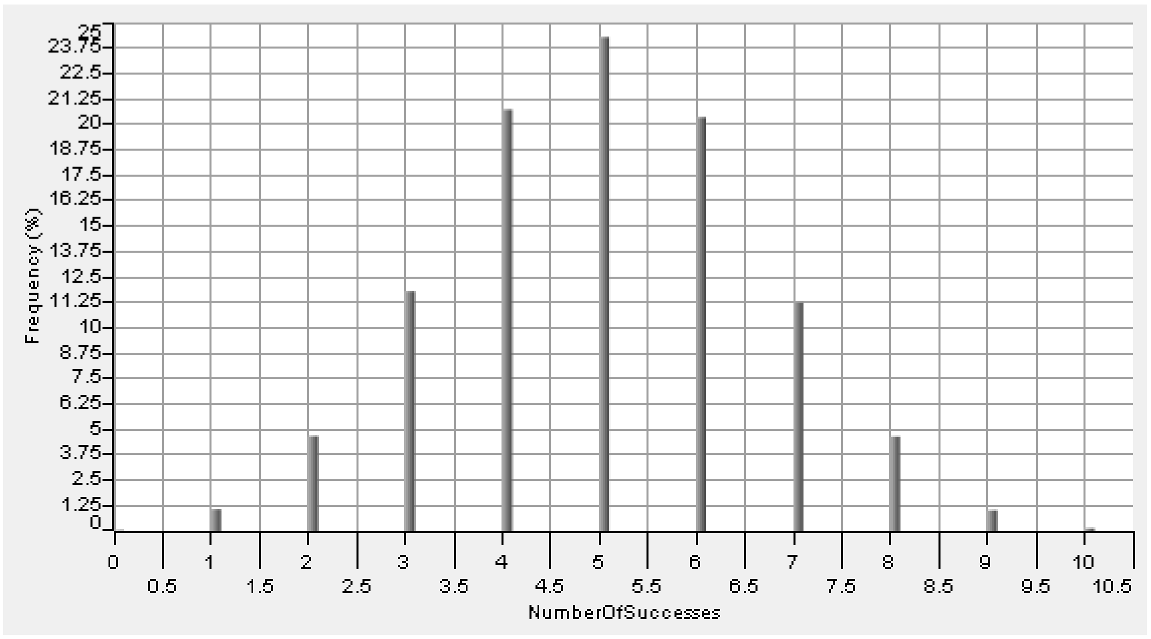
```
sample: (success success success failure success ... success failure)
```

If you want to see a histogram of this binomial distribution with success probability of 0.5, you can generate a large number of trials like this:

```
COPY 10000 numberOfTrials
REPEAT numberOfTrials
   BINOMIALSAMPLE 10 0.5 sample  '<== This line invokes the subroutine
   COUNT sample = success successCount
   SCORE successCount NumberOfSuccesses
END
HISTOGRAM binsize 0.1 percent NumberOfSuccesses
```

Here is the result of the above program:

[4]The *Statistics101* installation directory contains a folder called *lib* that contains this subroutine in a file called “binomialCommands.txt”. There is another file in the same directory named “multinomialCommands.txt”. That file provides commands (subroutines) that use this method to implement the general MULTINOMIAL distribution of which the binomial distribution is a special case. You might find it instructive to examine those subroutines.



If you now wanted to compute the probability of, say, exactly four successes out of the 10 samples, you could do it with this:

```
COUNT NumberOfSuccesses = 4 trialsWith4Successes
LET probability = trialsWith4Successes / numberOfTrials
PRINT probability
```

To get an answer of:

```
probability: 0.22
```

This last result can be computed mathematically without simulation by using the built-in BINOMIALPROB command:

```
BINOMIALPROB 10 0.5 4 probability
PRINT probability
```

Yielding a more precise answer:

```
probability: 0.205078125
```

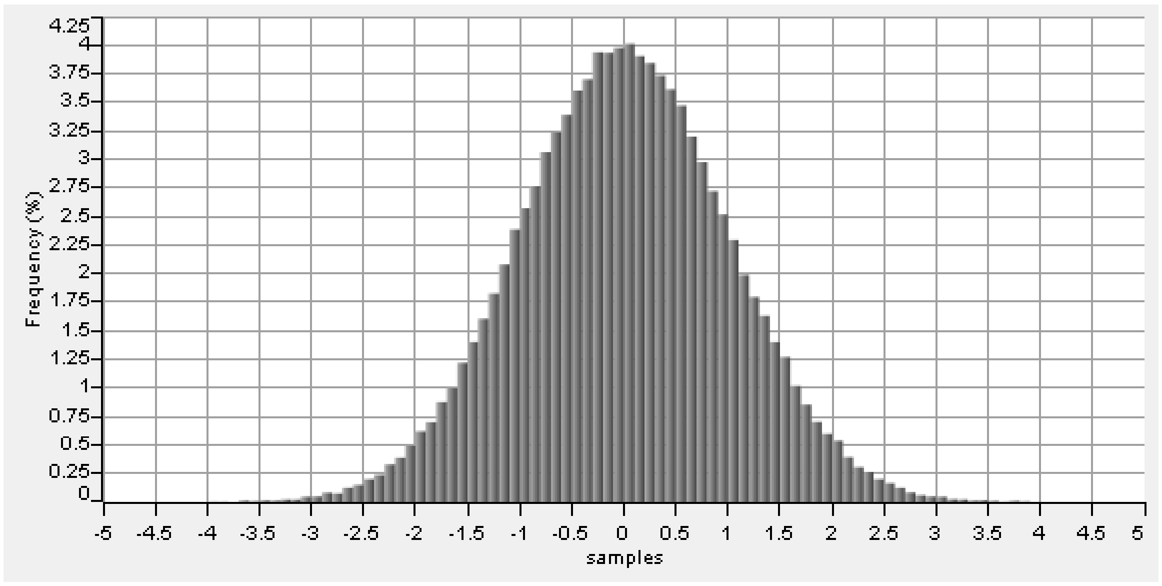
There is a lot more to learn about subroutines as you will see in the second part of this book, but what we have just covered is sufficient for you to understand any of the other uses of subroutines in this part.

Normal Distribution

In this section, we discuss the Normal distribution from the viewpoint of simulation and *Statistics101*. You are probably familiar with the bell-shaped “normal” or “Gaussian” curve. The normal distribution is a continuous distribution, which means, roughly, that its random variable has an unlimited number of values. You have probably used tables to determine probabilities from the normal curve. If so, you had to standardize your values using “z-scores” because one table can only represent one pair of mean and standard deviation. Since *Statistics101* is primarily a simulation system, how can you use simulation to determine probabilities of ranges of events on the normal curve? In *Statistics101* the NORMAL command fills a vector with a chosen number of “random” numbers derived from a normal distribution with a given mean and standard deviation. You can see this with the following two commands, the first of which generates a large number of numbers drawn from a standardized (mean = 0, standard deviation = 1) normal distribution, and the second then creates a histogram of those numbers.

```
NORMAL 100000 0 1 samples
HISTOGRAM percent binsize 0.1 samples
```

The above two-line program outputs this histogram:



This should look familiar. Now, if you wanted to know what the probability is that a number drawn from that standard distribution is within one standard deviation of the mean, here is all you need to do (repeating the above two command lines for completeness):

```
COPY 100000 numberOfSamples
NORMAL numberOfSamples 0 1 samples
COUNT samples between -1 1 samplesInRange
LET probability = samplesInRange / numberOfSamples
PRINT probability
```

Which as expected produces the familiar probability,

```
probability: 0.68352
```

In *Resampling Stats* it is not necessary to use a standardized normal curve. If you know that your mean is 100 and your standard deviation is 10, and you wanted to know what the probability is of values from that distribution being between 75 and 90, here’s a program that will produce the answer directly:

```
COPY 10000 numberOfSamples
NORMAL numberOfSamples 100 10 samples
COUNT samples between 75 90 samplesInRange
LET probability = samplesInRange / numberOfSamples
PRINT probability
```

This yielded the following answer:

```
probability: 0.1536
```

If you wanted to know the probability that the values are in the lower tail up through 75, you only need change the COUNT command to this:

```
COUNT samples <= 75 samplesInRange
```

Or to compute the probability that a number will be in the upper tail, say 90 and above, you would change it to this:

```
COUNT samples >= 90 samplesInRange
```

Here is a subroutine to compute the probability that a normal random variable's value will be in a given range using the technique shown in the above examples.

```
NEWCMD NORMALPR_RANGE mean stdDev lowLimit highLimit prob
   COPY 10000 numberOfSamples
   NORMAL numberOfSamples mean stdDev samples
   COUNT samples between  lowLimit highLimit samplesInRange
   LET prob = samplesInRange / numberOfSamples
END
```

This lets you perform the same computation as before, but using the subroutine, it takes only one line, like this:

```
NORMALPR_RANGE 100 10 75 90 probability
PRINT probability
```

You can write the other two subroutines (suggested names, NORMALPR_LOWERTAIL and NORMALPR_UPPERTAIL) using the different COUNT commands as an exercise[5].

Sampling Distribution of a Statistic

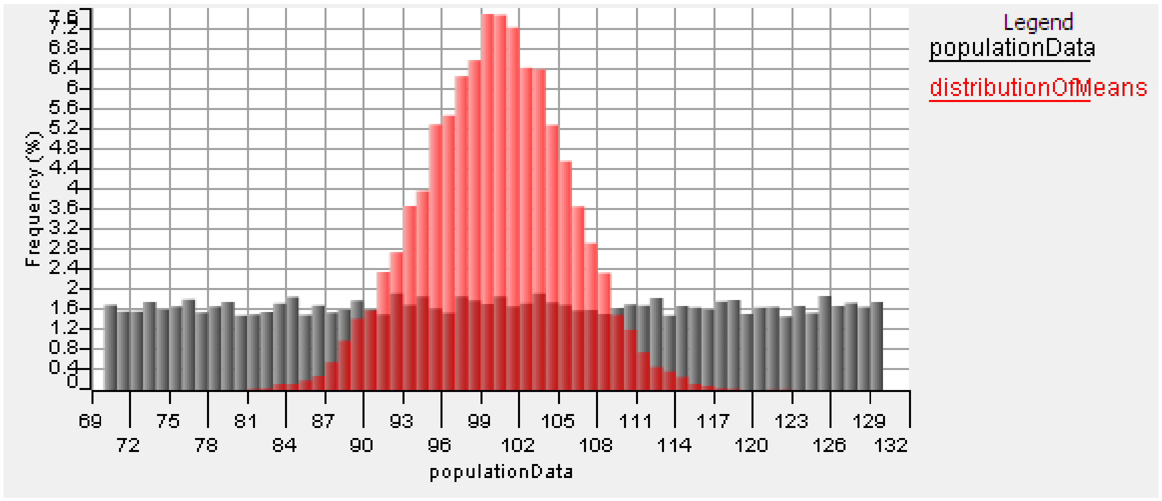
Suppose that you take a random sample from a population and compute the sample's mean. You would expect, since the sample did not include every member of the population, that the mean you calculated would differ from the population's mean. If you repeatedly took additional samples of the same size from the population and calculated their means, you would find that some samples would have means that were below the population mean and some would have means that were above it. If you then plotted all those sample means on a histogram, you would have a graph that illustrates the "sampling distribution of the means" or just "distribution of the means." Here's a program that performs those steps:

```
COPY 70 minValue   'lower limit of uniform distribution
COPY 130 maxValue  'upper limit of uniform distribution
COPY 10 sampleSize

COPY 10000 populationSize
COPY 10000 repeatCount
UNIFORM populationSize minValue maxValue populationData  'The original
population
REPEAT repeatCount
   SAMPLE sampleSize populationData sampleData  'One sample from original pop
   MEAN sampleData dataMean                     'Mean of sample
   SCORE dataMean distributionOfMeans
END
HISTOGRAM percent binsize 1 populationData distributionOfMeans
```

[5]You can use the built-in NORMALPROB command to precisely compute Normal probabilities.

The program starts by creating a population that has a uniform distribution of values between 70 and 130, which makes the population mean 100. These numbers were chosen arbitrarily, just for the sake of this example. Next, the program repeatedly takes samples of size 10 from that population, computes the mean, and accumulates the means in the vector `distributionOfMeans`. Finally, it displays a histogram like this one:



The gray bars are the histogram of the population. The red bars are the histogram of the means of all the samples of size ten. You can see that the center of the distribution of the means is very close to 100, as you would expect. You can also see that this distribution of the means looks much like a normal distribution. In fact, it becomes more and more like a normal distribution as the sample size is increased, per the Central Limit theorem.

The chart tells you that if you take a sample of ten from this population, the mean of the sample can vary from about 81 to about 122. If you did not know the population's mean and you took a sample of 10 to estimate it, then you might get an estimate that is inaccurate by as much as 22 (I.e., 22 = 122 – 100). Most of the time the estimate will be much closer to the correct value, but you would never know based on one sample of ten how far that sample's mean was from the correct value. I will discuss this issue further in the section Confidence Intervals.

You can use a similar process to generate the sampling distribution of any statistic, not just the mean. An easy statistic to try is the median: just substitute MEDIAN for MEAN in the above program and change the name of the scoring variable from `distributionOfMeans` (two places) to `distributionOfMedian` or, to be more general, `distributionOfStatistic`. This method (resampling) of generating the sampling distribution of a statistic can be used for a statistic of any complexity, and is especially useful with statistics whose sampling distributions have no closed mathematical or tabulated formulation.

You can experiment with the program by varying the sample size. You will find that a larger sample will lead to a narrower distribution of the means. That is, the distribution of the means for a larger sample will have a smaller standard deviation.

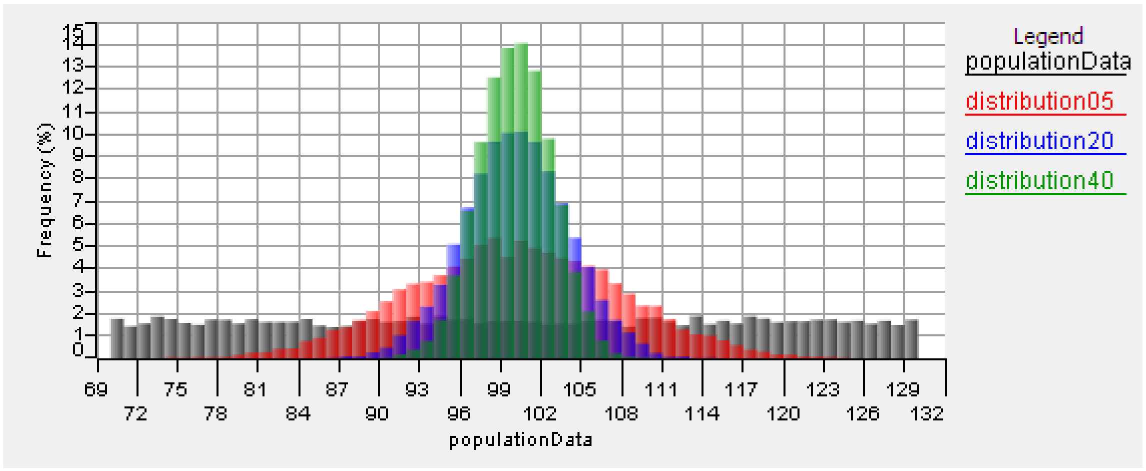
Here's a modification of the above program that will produce distributions for three different sample sizes, using a subroutine. The program plots all three distributions on the same histogram so you can visually compare the results.

```
'Subroutine to compute a vector containing the means of 10000 samples
'of size sampleSize taken from the given population data.
NEWCMD MEANDISTRIBUTION populationData sampleSize distributionOfMeans
   LET repeatCount = 10000
   REPEAT repeatCount
      SAMPLE sampleSize populationData sampleData
      MEAN sampleData dataMean
      SCORE dataMean distributionOfMeans
   END
END

COPY 70 minValue 'lower limit of uniform distribution
COPY 130 maxValue 'upper limit of uniform distribution
COPY 10000 populationSize
UNIFORM populationSize minValue maxValue populationData  'The original
population

MEANDISTRIBUTION populationData  5 distribution05  'sample size = 5
MEANDISTRIBUTION populationData 20 distribution20  'sample size = 20
MEANDISTRIBUTION populationData 40 distribution40  'sample size = 40
HISTOGRAM percent binsize 1 populationData distribution05 \
  distribution20 distribution40
```

Here is the histogram resulting from one run of the above program. You can clearly see that a larger sample size leads to a narrower distribution of the mean:



Knowing or simulating the sampling distribution of a statistic is what allows you to estimate confidence intervals and to estimate probabilities for “hypothesis tests.” I will discuss confidence intervals and hypothesis tests later in the sections with those names.

Statistics Examples

In statistics problems, you know the sample; you want to use your knowledge of the sample to tell you something about the population. Typical examples from real life are the never-ending political polls, or the television ratings measurements.

In the problems we have looked at so far, we had a well-defined population from which we took random samples in a well-defined way, scoring the samples on whether they satisfied our success criterion and averaging the results to get a probability. But what if all you had was a single sample and you wanted to know something about the unknown population that it came from?

What if someone presented you with a card deck and you drew two cards at random and found that they were aces? Would you be able to conclude that the deck was "stacked" or that it might not be a poker deck? What if you flipped a coin four times and came up with three heads. Could you say that the coin is not fair?

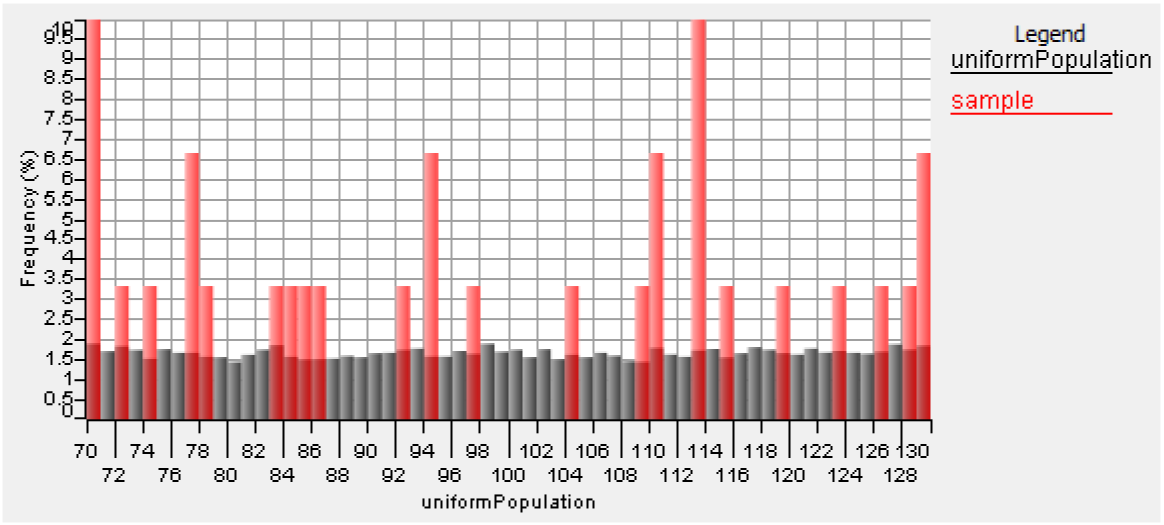
It should be clear from a thoughtful consideration of the above questions that a sample doesn't necessarily give you any certainty about the nature of the population. The same sample could have come from many different populations.

Would it be in error to conclude, for example, having thrown three heads out of four, one time, that the coin is unbalanced? You saw above that even if the coin is fair, one-fourth of the time you will throw three heads out of four. So you would have to reason something like this: "A 0.25 probability is really not very low, so the fact that I threw three out of four heads this time is not too surprising, even if the coin is fair. Therefore, I cannot conclude that the coin is unfair. Of course, I can't conclude for certain that the coin is fair, either. In fact, I wouldn't be surprised by either possibility based on this sample and the associated probabilities. If I want more certainty, I would need a bigger sample, one for which the probability of coming from a fair coin is either very high or very low, depending on which premise (fair or unfair) I want to test. How big should that sample be?"

The Sample versus the Population

The reason that any statistical sampling technique works is that a *random* sample of a population shares characteristics with the population. Statistical techniques take advantage of those similarities. Here are some examples that illustrate the similarities. First is a sample from a uniform distribution:

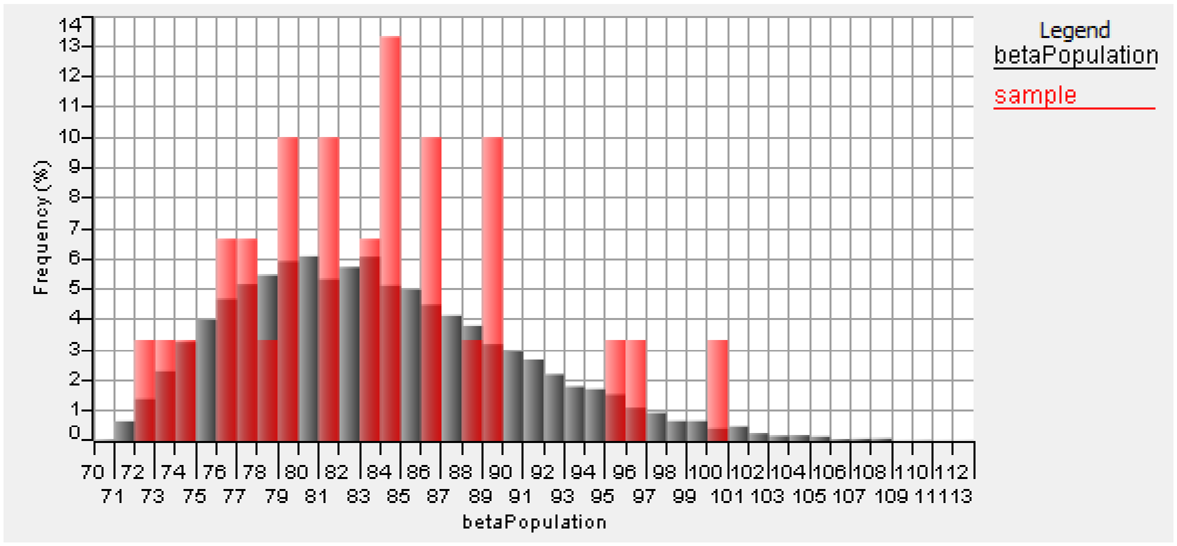
```
UNIFORM 10000 70 130 uniformPopulation
SAMPLE 30 uniformPopulation sample
HISTOGRAM percent binsize 1 uniformPopulation sample
```



You can see that the histogram of the population (in gray) is evenly distributed, as a uniform distribution should be, and the sample (in red) is also fairly evenly distributed over the population as expected. There are a few sample bins that are higher than the others, but even those higher bins are fairly evenly distributed and would tend to balance out if you were to calculate, say, a mean from the sample.

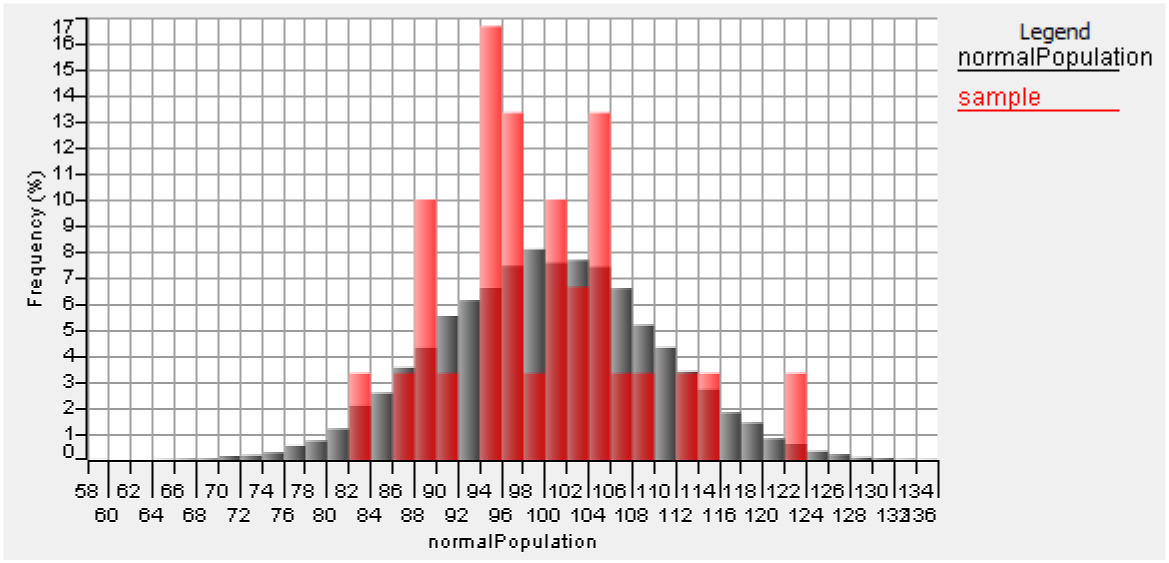
As another example, look at a beta distribution. The beta distribution, which is implemented as a subroutine, is non-symmetrical:

```
INCLUDE "lib/BetaDistribution.txt"
BETA 10000 70 130 3 10 betaPopulation
SAMPLE 30 betaPopulation sample
HISTOGRAM percent binsize 1 betaPopulation sample
```



Again, you can see that the sample is distributed similarly to that of the original population. Finally, for a normal distribution:

```
NORMAL 10000 100 10 normalPopulation
SAMPLE 30 normalPopulation sample
HISTOGRAM percent binsize 2 normalPopulation sample
```



Once again, the random sample resembles shape of the population's distribution. The resemblance will never be perfect, and as you would expect, it will vary from sample to sample. The larger the sample, the better it will resemble the population from which it is drawn.

Bias

A sample that is not random is called a *biased* sample. The above examples make clear that a random sample is a fair representation of the population because each member of the population has an equal chance of being selected. In any honest statistical study, much effort is spent on ensuring that the sample will be unbiased. If it is impossible to obtain an unbiased sample, then the study must discuss and explain the effect that the bias may have on the results, and the limitations on the conclusions because of the bias. There are many sources of bias. Here are a few common categories of bias.

Selection bias is the result of an improper method of choosing a sample. An Internet survey will be inherently biased because the responders choose themselves and are most likely to be those who are more strongly motivated one way or another about the subject than the population at large.

Survivor bias is the result of sampling only the survivors of a population to represent the whole population. For example, a study of the average performance of mutual fund companies over time will usually exclude those mutual funds that went out of business during that time. This biases the result to make it look like the overall performance was better than it actually was.

Reporting bias is the result of the fact that popular magazines and even scientific journals will tend to report surprising or unexpected or only positive results. If twenty studies were done to determine the effects of blueberries on skin cancer and 19 of the studies showed no improvement, but one showed improvement, then it is very likely that the one anomalous study will be published in the scientific press and hyped in the popular press, while the others will be ignored. With the 5% level of significance that is used most frequently for studies, 1 out of 20 repetitions of a study will be anomalous just by chance. And that's the one that will be published.

Data mining bias is the result of re-analyzing the data to find unanticipated correlations. For example, analyzing a given set of stock market data for different patterns of price movement, you might find that one particular pattern of price movement, if traded a certain way, gives remarkable profits. But although the pattern might work for the given data, it would be naive (and expensive) to believe that it will work on future movements.

Tip: A good book for the general reader, which discusses the pervasiveness of many kinds of bias in published studies, is Wrong: Why experts keep failing us--and how to know when not to trust them by David H. Freedman. The book has no mathematics and will open your eyes to the weakness of many of the studies that are reported in the press and act as a warning for your own statistical investigations.

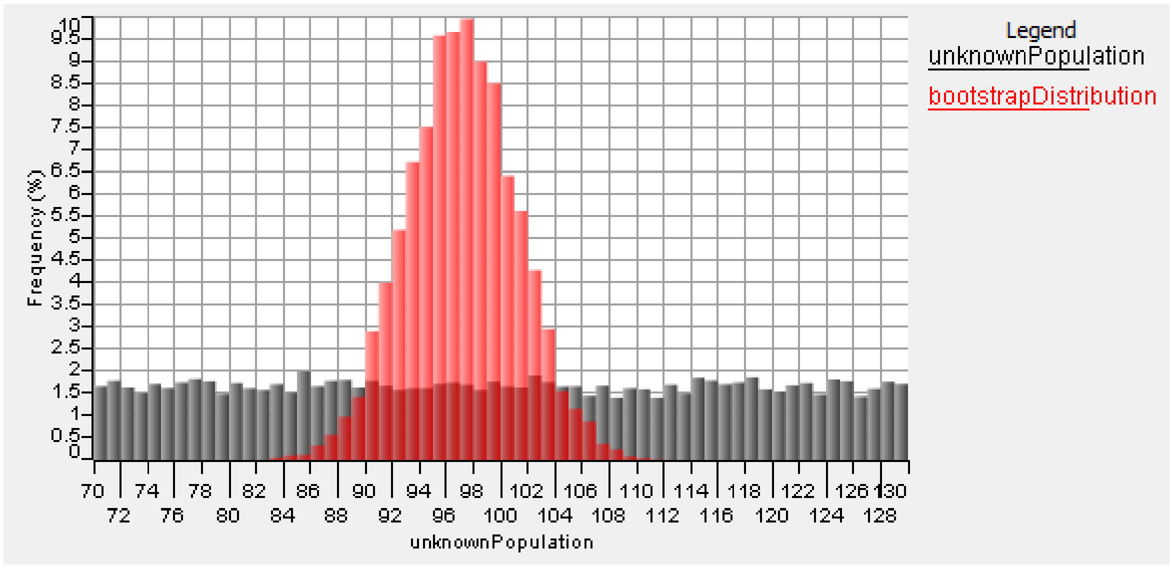
The Bootstrap

The reason we take samples in the real world is because we want to know something about the population even though we cannot survey the entire population because of cost, accessibility, or some other reason. As we have just seen, a random sample is representative of the population, so if the population is otherwise unknown, what would happen if we turn our logic around and say that the sample *is* the population? That is, what if we replicate our sample millions of times to construct an artificial population from which we can draw new samples? *Instead of drawing the sample from the population, we would be drawing the population from the sample*. This sounds like getting something for nothing, like magic, or like lifting yourself by pulling on your bootstraps. It's no surprise therefore that this idea has been named "the bootstrap."

A common use for the bootstrap is to compute the distribution of a statistic such as the mean. This is done as follows. Starting with your randomly sampled data series, you repeatedly create a new "bootstrap" sample *of the same size* as the original sample by randomly drawing values, *with replacement*, from your original sample data. Sampling with replacement is equivalent to sampling from an infinite number of copies of each member of the sample. For each of these new bootstrap samples, you compute the desired statistic (such as mean, median, standard deviation, or any other) and save it in a vector, which upon completion of the desired number of repetitions, will contain an empirical estimate of the sampling distribution of that statistic.

To see how this works, suppose you took a sample from a population whose mean you wanted to know and then used the bootstrap method to generate a distribution of the mean. The reason for being interested in the distribution of the mean or the distribution of any other statistic is that the distribution is needed to evaluate "hypothesis tests" and "confidence intervals." These will be discussed later. For this example, we'll take a sample from a known population and pretend we don't know anything about the population.

```
UNIFORM 10000 70 130 unknownPopulation      'The "unknown" population
SAMPLE 30 unknownPopulation originalSample  'Take the sample
REPEAT 10000
   SAMPLE 20 originalSample bootstrapSample 'Resample to get bootstrap sample
   MEAN bootstrapSample bootstrapSampleMean 'Compute the desired statistic
   SCORE bootstrapSampleMean bootstrapDistribution 'Save statistic in vector
END
HISTOGRAM percent binsize 1 unknownPopulation bootstrapDistribution
```



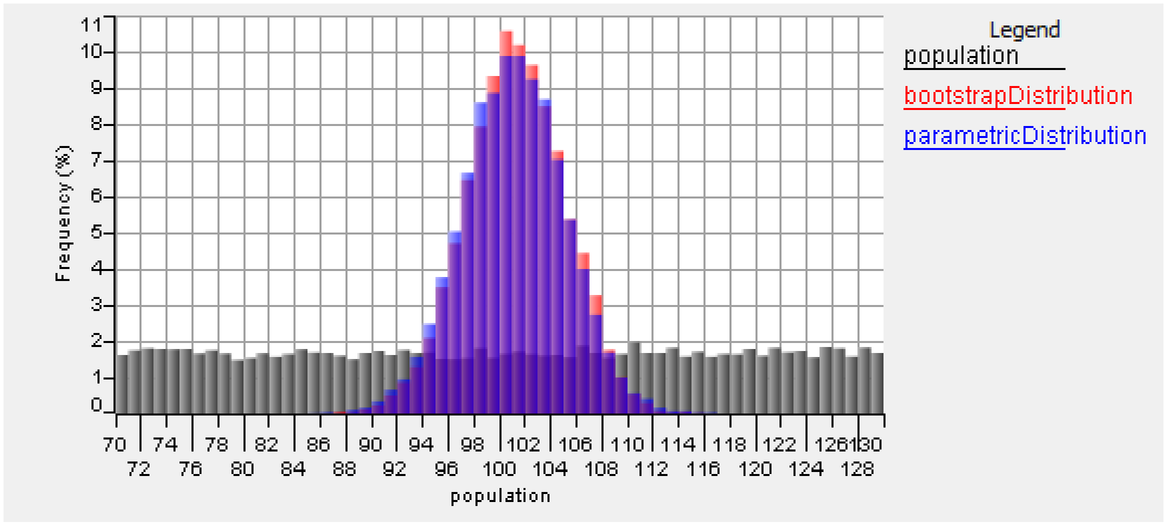
Notice that the actual mean of the population is 100 but that the center of the bootstrap distribution is slightly lower. That is because the distribution was generated from a single original sample whose mean was not exactly that of the population. It would be rare that any sample's mean would be exactly the same as its population's mean, but it would also be rare that it would be radically different. That is what the distribution of the means tells us: the variation from the population's mean that we might expect in a sample's mean.

The traditional way to generate the distribution of means is to calculate the mean and standard deviation of the sample, use those as characterizing a normal curve that is then taken as the population. Then, the mathematical properties of the normal curve are used to derive the distribution of the means.

The next program will compare the results of the bootstrap method and the traditional, or "parametric" method:

```
sampleSize = 20
UNIFORM 10000 70 130 population     'The "unknown" population
SAMPLE 30 population originalSample 'Take the sample
MEAN originalSample originalMean    'Find the sample's mean
STDEV originalSample originalStdDev 'Find its standard deviation
'Create a normal population from the sample's mean and Std Dev:
NORMAL 10000 originalMean originalStdDev parametricPopulation
'Take samples from both populations and save all the means
REPEAT 10000
   SAMPLE sampleSize originalSample bootstrapSample
   MEAN bootstrapSample bootstrapSampleMean
   SCORE bootstrapSampleMean bootstrapDistribution

   SAMPLE sampleSize parametricPopulation parametricSample
   MEAN parametricSample parametricSampleMean
   SCORE parametricSampleMean parametricDistribution
END
HISTOGRAM percent binsize 1 population bootstrapDistribution \
  parametricDistribution
```



You can see that the two methods give very similar results. Note also that the two methods differ only in the way the artificial population was generated. The parametric method used the mean and standard deviation of the sample to generate a population based on a normal distribution and then sampled that population to generate the distribution of means. The assumption of a normal distribution is false in this case, but as you can see, it didn't make much difference. The bootstrap method used the actual data of the sample to generate the population, without making any assumptions as to the shape of the population's distribution. The parametric method would normally not be completed by simulation as shown here. Instead, further mathematical steps would be taken, such as using Student's t distribution method with tables.

Since you will most often use bootstrapping to generate the distribution of means, here's a subroutine that will do the work. This subroutine is included in the library file "lib/bootstrapCommands.txt" along with two others that resample the median and the standard deviation to obtain their respective sampling distributions. You can see an example usage of BOOTSTRAPMEAN in the next section.

```
' BOOTSTRAPMEAN resamples a vector containing sample data
' computing the mean a given number of times to produce a
' distribution of the means.
' Inputs:
'    sampleData: the random sample taken from the population.
'    numberOfMeans: the number of resampled means to be computed.
' Output:
'    distributionOfMeans: vector containing all the means that were
'     calculated from resamples of the sampleData vector.
NEWCMD BOOTSTRAPMEAN sampleData numberOfMeans distributionOfMeans
   CLEAR distributionOfMeans
   SIZE sampleData sampleSize
   REPEAT numberOfMeans
      SAMPLE sampleSize sampleData  bootSample
      MEAN bootSample bootSampleMean
      SCORE bootSampleMean distributionOfMeans
   END
END
```

However strange "bootstrapping" may sound, it is conceptually little different from computing a mean and a standard deviation from a sample and using those to draw conclusions about an otherwise unknown population. Both are characterizing the unknown population based on the sample. One important difference is that bootstrapping from a sample does not assume that the population is normal, which the parametric approach does.

Question for discussion:

A sampling distribution of the means generated by bootstrap from a sample has the limitation that the lowest possible value of the mean is equal to the lowest value of the sample and the highest value of the mean is equal to the highest value in the sample. By contrast, the ideal distribution of means is a normal distribution that has plus and minus infinity as its extremes. Is this difference a practical problem with the bootstrap method?

Next, we'll show how to use the bootstrap to determine confidence intervals and following that, how to use it to perform hypothesis tests. A *confidence interval* is used to answer a question about the accuracy of a parameter of the population based upon sample evidence. A *hypothesis test* is used to discover how likely or unlikely it is for a particular sample to have come by chance from a particular population.[6]

[6] After Simon, *Resampling: The New Statistics*, page 298.

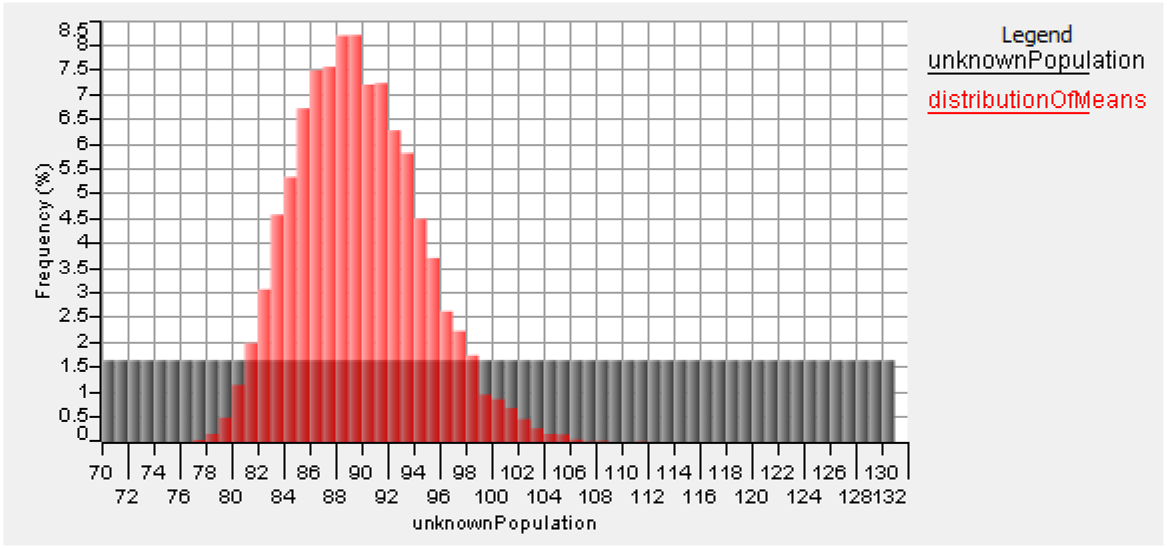
Confidence Intervals

Given a population and a sample size, the distribution of the mean describes the probability that a sample mean will be a certain distance from the population mean. A confidence interval goes in the reverse direction: given a sample, it estimates the probability that the mean (or any other chosen parameter) of the unknown population from which the sample came will be within a certain distance from the sample's mean (or other statistic).

As mentioned earlier, when you take a sample, the sample's mean will at best be an approximation of the population's mean. If you were unlucky, your sample's mean might be quite a bit off the population's mean. So you want to find the distribution of the means to get an idea as to how far off your estimate might be. Let's demonstrate what happens when you are unlucky, that is, you happened unknowingly to get a sample that is quite far from the population mean. The next program starts off with a sample that was taken from a uniform distribution of integers between 70 and 130. The sample's mean is 89.7, which is far off the population mean of 100.

```
COPY 70,130 unknownPopulation
DATA (87.0 104.0 76.0 97.0 127.0 81.0 74.0 81.0 82.0 88.0) sampleData
MEAN sampleData sampleDataMean
PRINT sampleDataMean
BOOTSTRAPMEAN sampleData 10000 distributionOfMeans
HISTOGRAM percent binsize 1 unknownPopulation distributionOfMeans
```

The program uses the BOOTSTRAPMEAN subroutine given earlier to determine the distribution of means. You can see in the following histogram that the distribution is centered around 90 and that the original population is centered on 100. Since the mean of a sample will only rarely be exactly equal to that of the population, we would like to have some way of measuring how far off our estimate of the population mean might be. That measure is the confidence interval.



A sample mean is as likely to be above the population mean as it is to be below it. We want to define an interval around our sample mean that we can be sure would include the population mean. A typical probability used to define a confidence interval is 0.95 or 95%. Therefore, in this example, we need to find the upper and lower values along the X axis between which would be found 95 percent of the sample means, which is the same as 95 percent of the area of the distribution, centered on the sample mean. Rather than using tables, we can use the PERCENTILE command to find those upper and lower limits. We want the interval to enclose the middle 95 percent. That leaves 5 percent for the upper and lower excluded portions of the distribution. Dividing that in half gives 2.5 percent for each excluded portion. Therefore, we'll ask for the 2.5th and the 97.5th (i.e., 100 - 2.5) percentiles:

```
PERCENTILE distributionOfMeans (2.5 97.5) confidenceInterval
PRINT confidenceInterval
```

With the result being:

```
confidenceInterval: (81.3 100.1)
```

That means that "we estimate that there's a 95% chance that the population's mean will be found between 81.3 and 100.1." You can see that in this extreme case, we just barely made it.

This next example is taken from *Statistics the Easy Way* by Douglas Downing and Jeffrey Clark, page188: Randomly select 20 counties in the United States and record their populations. Use that sample to estimate, within a 95% confidence interval, the mean population for all counties in the US. Here, a 95% confidence interval would be the range of county populations that would have a 0.95 probability of containing the mean value for the whole U.S. Don't confuse the two meanings of "population" here. One is the count of people in a county and the other is the statistical population, which in this case is the set of counties.

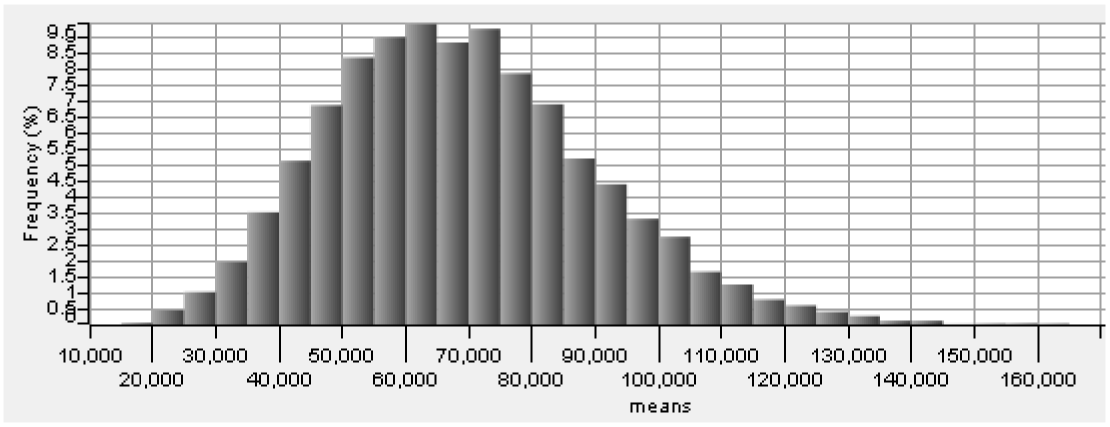
Leaving out the county names, here are the populations of 20 counties randomly selected from the 1990 census (the mean of these numbers is 68,791):

30083	152585	50917	300836
98928	6179	93145	32273
460	16754	26147	14194
11355	76779	7129	7021
32343	352910	51150	14632

Here's a *Resampling Stats* program that uses the bootstrap technique to answer the question. You can cut and paste this into *Statistics101* and run it. I used the DATA command, which is a synonym for the COPY command, to emphasize that the `countySample` vector is the program's input data. I could have used the BOOTSTRAPMEAN subroutine to do the work that is being done in the REPEAT loop, but I chose not to so that all the details of the process would be visible.

```
DATA (30083 152585 50917 300836 98928 6179 93145 \
   32273 460 16754 26147 14194 11355 76779 7129 7021 \
   32343 352910 51150 14632) countySample
REPEAT 10000
   SAMPLE 20 countySample newSample
   MEAN newSample newSampleMean
   SCORE newSampleMean means
END
HISTOGRAM percent binsize 5000 means
PERCENTILE means (2.5 97.5) confidenceInterval
PRINT confidenceInterval
```

The program repeatedly takes 20 random samples, with replacement, from the initial bootstrap sample, computes the mean, and saves the mean in a vector. After the final pass through the loop, the vector `means` contains the 10000 means from the 10000 samples of 20 values from the bootstrap sample `countySample`. That vector comprises the distribution of the sample means, which is made visible by the HISTOGRAM command, and looks like this.



There is a lot of variation among these means and somewhere in the range of that variation is the mean of the unknown population. The program computes the values for the 2.5th percentile and the 97.5th percentile. That is the range within which one can expect, with a 95% probability, to find the mean of the population from which the sample was drawn.

Here are the results of two runs of the program (without the histograms):

```
confidenceInterval: (32255.85  115508.85)
confidenceInterval: (32108.375 114477.125)
```

The interval that Downing and Clark computed using the t-distribution technique was (23505.8 114076.2). The bootstrap answer gave a slightly narrower interval. The total number of counties and the total population of the US are known, so you can actually calculate the mean population for the counties. For the census from which the data were taken there were 248,709,873 people divided by 3,135 counties (including independent cities). Therefore the mean is 79,333.

Downing and Clark go on to say, "The main difficulty here is that the distribution of county populations does not follow a normal distribution. There are a few very large counties that raise the overall population average, but there is a very good chance that none of the large counties will appear in a [random] sample of this size. A larger sample would make the distribution of the sample average be closer to a normal distribution (according to the central limit theorem) and it would allow you to obtain a narrower confidence interval." Since the bootstrap approach makes no assumption of normality but the t-distribution method does, it is not surprising that there is some difference between the two answers.

Here is another example of computing confidence intervals. A poll was taken before an election in which three politicians were competing. The sample size was 400. The result was that 10 percent of those polled favored politician A, 40 percent favored politician B, and the rest favored politician C. What are the 95 percent confidence intervals for the results for each politician?

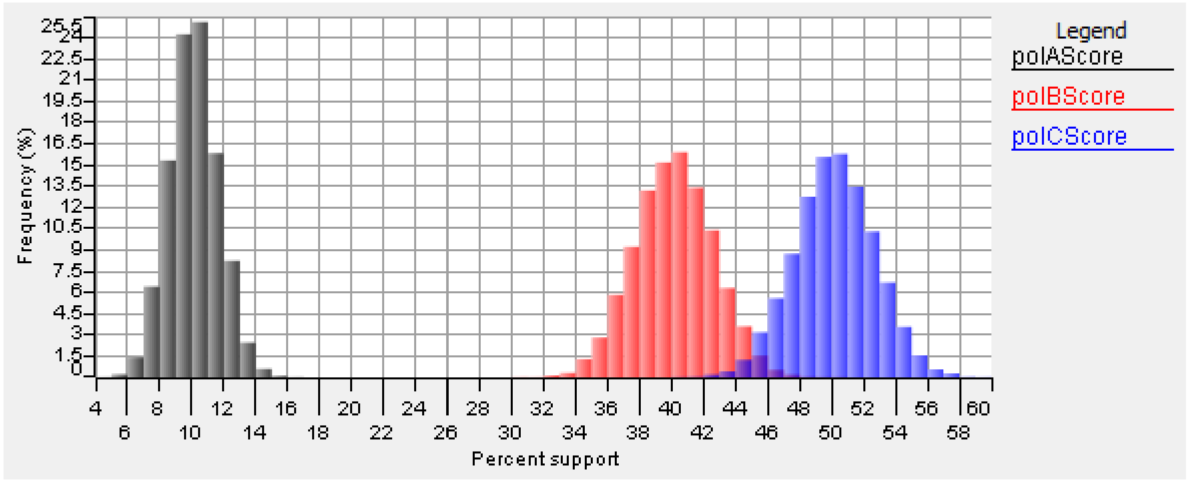
For the solution to this, you would construct a population that has the stated percentages of partisans for each politician, and then repeatedly take 400 samples from that population while recording the percentages that appear in each sample. This process will estimate the sampling distribution of the statistic of interest, the percentage support for each politician. You would then analyze the sampling distributions to find the confidence intervals. The following program will solve the problem.

```
ENUM polA polB polC
'Construct a population model based on the sample results:
COPY 10#polA 40#polB 50#PolC population
COPY 10000 repeatCount
COPY 400 sampleSize
REPEAT repeatCount
   SAMPLE sampleSize population sample
   COUNT sample = polA polACount
   COUNT sample = polB polBCount
   COUNT sample = polC polCCount
   polAPercent = polACount / sampleSize * 100
   polBPercent = polBCount / sampleSize * 100
   polCPercent = polCCount / sampleSize * 100
   SCORE polAPercent polAScore
   SCORE polBPercent polBScore
   SCORE polCPercent polCScore
END
PERCENTILE polAScore (2.5 97.5) polAConfidence
PERCENTILE polBScore (2.5 97.5) polBConfidence
PERCENTILE polCScore (2.5 97.5) polCConfidence
HISTOGRAM percent binsize 1 "" "Percent support" polAScore polBScore \
  polCScore
PRINT sampleSize polAConfidence polBConfidence polCConfidence
```

Running this produced the following confidence intervals...

```
polAConfidence: (7.25 13.0)
polBConfidence: (35.375 45.0)
polCConfidence: (45.25 54.75)
```

...and the following histogram:



Question for discussion:

Note the interesting fact that the confidence interval for the politician with the lowest following is narrower than the others. Why is that the case?

Read http://www.resample.com/content/text/26-Chap-22.pdf for an explanation.

Summarizing, the way to find a confidence interval given a set of sample data is to re-sample the original sample data, with replacement, as if it were the population. Your resampling sample size should be the same size as the original sample. Next, compute the measurement (the "statistic") you are interested in from the new sample. In the previous examples, the measurement was the mean or a percentage, but it can be any measurement that is appropriate to the problem you are analyzing. Then record the measurement in a scoring vector using the SCORE command. Repeat the resampling, the measurement, and the recording of the measurement some large number of times. Finally, use the PERCENTILE command to determine the limits of your confidence interval.

Hypothesis Testing

In hypothesis testing, you take a question like "is the coin fair?" and consider the possible "universes" from which the coin might have come. A coin can be unfair in many ways, but fair in only one. So you could guess that the coin has unfair odds of 3:2 or 3:4, etc, and choose one of those as a "benchmark" or "null" hypothesis to test whether our sample could have come from it. But there is no particular reason to choose one over another of these. And if you did choose one of them, say 3:2, and concluded that our sample had a very low probability of coming from that universe, you still haven't answered the question of whether the coin is fair. You want to choose a unique universe in which membership or non-membership of our sample will answer the fairness question. By "membership," I mean that there is a high probability that the sample could have come from the chosen universe; by "non-membership" I mean there is a very low probability that the sample could have come from the universe. The question of fairness can be most easily determined by choosing a "fair," 1:1, universe as our benchmark or "null" hypothesis and seeing whether our sample is likely to have come from such a universe. If not, then you know the coin is unfair, you just don't know by how much. If yes, you can say that the coin is probably fair.

The above reasoning is an informal example of the idea of "hypothesis testing." More formally, the process of hypothesis testing follows this pattern:

Choose a "research hypothesis" that states the expectation that you want to test. (e.g., "The coin is unfair.") This is essentially the hypothesis that the sample came from such-and-such a universe (in this case an unspecified "unfair" universe).

Derive the "null hypothesis" or "benchmark hypothesis," which is the one you will actually test. The null hypothesis is derived from the research hypothesis by negation. (e.g., "The coin is fair.")

Decide on a probability or a range of probabilities outside which you will consider the null hypothesis to be rejected. (e.g., "95 percent probability").

Test the null hypothesis using appropriate means. This test is the calculation of the probability that the sample could have come from the universe of the null hypothesis based on some statistic. The statistic used should be chosen carefully. Common statistics are the sample mean, the variance, the sum of squared deviations, and so forth. Each statistic has a probability distribution called its sampling distribution. You find where your sample falls on that distribution to calculate the probability that the sample satisfies the null hypothesis. Traditionally, you would select a formula or a table to obtain the answer. Which formula, which table to use? That might not be evident in a particular case, especially to a beginner. In this paper, we don't use formulas and tables. We use the resampling method to solve all the problems we encounter.

If the probability that the sample could have come from the null hypothesis universe is outside the pre-determined region of acceptance, you would conclude that the null hypothesis is rejected. If the null hypothesis is rejected, you can say that the research hypothesis is "supported" (you can't say it is "true," or "confirmed"). What you could say in truth is that "if the research hypothesis turns out to be true, you won't be surprised."

If the sample could have come from the benchmark universe, based on your range of acceptance, then the null hypothesis is not rejected and the research hypothesis is rejected.

The hypothesis test resembles a proof by contradiction. You make an assumption, the null hypothesis ("the sample came from a fair coin"), trace it to its conclusion ("after a large number of trials a sample similar to our original sample occurred only X percent of the time, which is very low") and find that the conclusion contradicts the assumption, therefore you reject the assumption. Since the research hypothesis was the opposite of the assumption you rejected, you consider the research hypothesis to have been supported. The hypothesis test is really not a proof, however. It is more a plausibility argument that the research hypothesis is valid.

Let's return to our example of the unfair coin. Say that you flipped the coin ten times and got seven heads. Is the coin fair? Your research hypothesis is that the coin is unfair. Therefore, the null hypothesis, which is the opposite of the research hypothesis, and which you are hoping to contradict, is that the coin is fair. You then choose a probability criterion which you consider so low that if the computed probability were equal to or less than the criterion then it would be reasonable to say that the sample could not have come from the benchmark universe. Say you choose 0.05. Then you would determine the probability of seven heads out of ten throws from that universe and compare it to your criterion. Here is a *Resampling Stats* program to perform that calculation.

```
ENUM heads tails
COPY (heads tails) coin
COPY 10000 numberOfTrials
REPEAT numberOfTrials
   SAMPLE 10 coin tenFlips
   COUNT tenFlips = heads headCount
   SCORE headCount result
END
COUNT result = 7 successes
LET probability = successes / numberOfTrials
PRINT probability
```

The result comes in around 0.12. Since this is greater than your criterion of 0.05, you must conclude that seven heads out of ten is not sufficiently unusual for a fair coin, so you cannot reject the null hypothesis.

Here is another example of hypothesis testing, this one taken from CliffsQuickReview Statistics p. 80, example 5: "A professor wants to know if her introductory statistics class has a good grasp of basic math. Six students are chosen at random from the class and given a math proficiency test. The professor wants the class to be able to score at least 70 on the test. The six students get scores of 62 92 75 68 83 95. Can the professor be at least 90 percent certain that the mean score for the class on the test would be at least 70?" The null hypothesis would therefore be: mean score < 70. The book solves the problem using the t distribution. Instead, we solve it using the bootstrap:

```
DATA (62 92 75 68 83 95) scores
COPY 10000 numTrials
COPY 0 successCount            'Establish a counter
REPEAT numTrials
   SAMPLE 6 scores sample       'Take a bootstrap sample
   MEAN sample sampleMean       'Compute the desired statistic
   IF sampleMean < 70           'Count it if it satisfies null hypothesis
      successCount = successCount + 1
   END
END
LET probability = successCount / numTrials 'Compute probability of success
PRINT probability
```

The result of running this was

```
probability: 0.0304
```

That number is the probability of the null hypothesis, which is less than 10 percent. Therefore, the professor can be at least 90 percent certain that the class score would be at least 70.

Here's a more complicated example using resampling to perform a hypothesis test. The example is taken from *Statistics the Easy Way* by Douglas Downing and Jeffrey Clark, (Chapter 18 exercise 5, page 223). Suppose four new pesticides are being tested in a laboratory, with the results shown in the following table, called a *contingency table*. Is pesticide 1 significantly better than the rest?

	Type 1	Type 2	Type 3	Type 4	Total
Insects killed	139	100	73	98	410
Insects surviving	15	50	80	47	192
Total tested	154	150	153	145	602

Following the six steps listed earlier, you decide that your research hypothesis is that "type 1 is significantly better than the rest." The null or benchmark hypothesis to be tested is that "type 1 is no better than the rest." Translating that to a statistical statement, you get, "The results of the type 1 pesticide come from the same population as those for the other pesticides." You choose the 5% level of significance for traditional reasons. Now, you need to decide on a statistic to use to compare different test results with one another. (This would normally be done with the chi-square test, which can also be done in *Statistics101* as will be demonstrated later. But just to show that the chi-square statistic is not the only alternative, I'll use a simpler one first.) One test that comes immediately to mind is to compare the ratio of survivors to total tested (in each sample) for the best and the worst pesticide. Using the ratios removes the influence of differing sample sizes, although here the samples are all very close to the same size. So the test becomes "what is the probability that the best and worst of four groups randomly selected from the benchmark population would by chance have a survival rate that differs by (47/145 - 15/154 = 0.227) or more?" Next, you need to define the benchmark population. The benchmark is that "out of 602 tested, 192 survived." So you define the benchmark population as consisting of 192 survivors and 410 fatalities.

Here's the simulation:

```
COPY 0.227 testRatioDifference
COPY 1000 numberOfTrials
ENUM died survived
DATA 410#died 192#survived nullPopulation
REPEAT numberOfTrials
   SAMPLE 154 nullPopulation type1
   SAMPLE 150 nullPopulation type2
   SAMPLE 153 nullPopulation type3
   SAMPLE 145 nullPopulation type4
   COUNT type1 =survived type1SurvivorCount
   COUNT type2 =survived type2SurvivorCount
   COUNT type3 =survived type3SurvivorCount
   COUNT type4 =survived type4SurvivorCount
   LET ratio1 = type1SurvivorCount / 154
   LET ratio2 = type2SurvivorCount / 150
   LET ratio3 = type3SurvivorCount / 153
   LET ratio4 = type4SurvivorCount / 145
   'Combine the ratios for easy comparison:
   COPY ratio1 ratio2 ratio3 ratio4 ratios
   MIN ratios minRatio
   MAX ratios maxRatio
   LET ratioDifference = maxRatio - minRatio
   SCORE ratioDifference distributionOfRatioDifference
END 'Show the sampling distribution:
HISTOGRAM binsize 0.01 distributionOfRatioDifference
COUNT distributionOfRatioDifference >= testRatioDifference successCount
LET probability = successCount / numberOfTrials
PRINT probability
```

The result of this simulation is:

```
probability: 0.0
```

Out of 1000 trials from the null population there were no cases as extreme as the observed difference between ratios. Since that is well below our critical value (0.05), you can reject the null hypothesis.

For comparison, the next program analyzes the same data using the chi-square technique. To use the chi-square method, you first must calculate the expected values for each cell in the data table based on the null universe. You do that by multiplying the cell's column's total by its row's total divided by the grand total. For example, the first cell's expected value would be 154 * 410 / 602 = 104.88. You do that for each cell as you can see in the next table, which is the same as the previous table except that the predicted frequencies are shown in parentheses.

	Type 1	Type 2	Type 3	Type 4	Total
Insects killed	139 (104.88)	100 (102.16)	73 (104.20)	98 (98.75)	410
Insects surviving	15 (49.12)	50 (47.84)	80 (48.80)	47 (46.25)	192
Total tested	154	150	153	145	602

In the program, you copy them into a `predictedFrequencies` vector in the same order as that for the observed values. You can see the `predictedFrequencies` vector in the program's first line.

```
DATA (104.88 102.16 104.20 98.75 49.12 47.84 48.80 46.25) \
predictedFrequencies
DATA (139 100 73 98 15 50 80 47) observedValues
CHISQUARE observedValues predictedFrequencies observedChiSquare
PRINT observedChiSquare
COPY 10000 numberOfTrials
ENUM died survived
COPY 410#died 192#survived nullPopulation
REPEAT numberOfTrials
   SAMPLE 154 nullPopulation type1
   SAMPLE 150 nullPopulation type2
   SAMPLE 153 nullPopulation type3
   SAMPLE 145 nullPopulation type4

   COUNT type1 =died type1DeathCount
   COUNT type2 =died type2DeathCount
   COUNT type3 =died type3DeathCount
   COUNT type4 =died type4DeathCount

   COUNT type1 =survived type1SurvivorCount
   COUNT type2 =survived type2SurvivorCount
   COUNT type3 =survived type3SurvivorCount
   COUNT type4 =survived type4SurvivorCount

   COPY type1DeathCount type2DeathCount type3DeathCount \
     type4DeathCount deathCounts
   COPY type1SurvivorCount type2SurvivorCount type3SurvivorCount \
     type4SurvivorCount survivorCounts
   COPY deathCounts survivorCounts observedSampleValues
   CHISQUARE observedSampleValues predictedFrequencies chiSquareStatValue
   SCORE chiSquareStatValue chiSquareDistribution
END
'Show the chi-square distribution:
HISTOGRAM binsize 0.5 chiSquareDistribution
COUNT chiSquareDistribution >= observedChiSquare successCount
LET probability = successCount / numberOfTrials
```

```
PRINT probability
OUTPUT "Null hypothesis is "
IF probability >= 0.05
   OUTPUT "NOT "
END
OUTPUT "rejected at the 5% significance level.\n"
```

Note that I added OUTPUT commands and an IF command at the end of the program so that it would automatically print the conclusion: acceptance or rejection of the null hypothesis. Here is the result from the above program:

```
observedChiSquare: 64.25130843346504
probability: 0.0
Null hypothesis is rejected.
```

We have achieved the same answer in both analyses. Granted, the first method, comparing the difference of ratios, is probably not as sensitive (“powerful”) as the chi-square method since it uses only two of the four groups (the ones with the max and the min ratios) whereas the chi-square method uses all four to arrive at its conclusion. Further, notice that I didn't have to use an F-distribution table, nor did I have to worry about how many "degrees of freedom" were involved. That's because our results vector "automatically" contained the data that constituted the sampling distribution appropriate to our sample. You can see the distribution by looking in the histogram produced by the HISTOGRAM command.

Although the program I have written solves the problem, I can use it as one more opportunity to demonstrate the use of subroutines. Studying the program reveals that several commands (SAMPLE and COUNT) are repeated together with different arguments. That is a sign that perhaps they can be made into a subroutine. Here (next page) is the same program rewritten using a subroutine.

```
ENUM died survived
GLOBAL died survived nullPopulation

NEWCMD COLUMN_TRIAL sampleSize deathCount survivorCount
   SAMPLE sampleSize nullPopulation columnSample
   COUNT columnSample =died deathCount
   COUNT columnSample =survived survivorCount
END

DATA (104.88 102.16 104.20 98.75 49.12 47.84 48.80 46.25)   \
  predictedFrequencies
DATA (139 100 73 98 15 50 80 47) observedValues
CHISQUARE observedValues predictedFrequencies observedChiSquare
PRINT observedChiSquare
COPY 1000 numberOfTrials

COPY 410#died 192#survived nullPopulation
REPEAT numberOfTrials
   COLUMN_TRIAL 154 type1DeathCount type1SurvivorCount
   COLUMN_TRIAL 150 type2DeathCount type2SurvivorCount
   COLUMN_TRIAL 153 type3DeathCount type3SurvivorCount
   COLUMN_TRIAL 145 type4DeathCount type4SurvivorCount
   COPY type1DeathCount type2DeathCount type3DeathCount \
     type4DeathCount deathCounts
   COPY type1SurvivorCount type2SurvivorCount type3SurvivorCount  \
     type4SurvivorCount survivorCounts
   COPY deathCounts survivorCounts observedSampleValues
   CHISQUARE observedSampleValues predictedFrequencies chiSquareStatValue
   SCORE chiSquareStatValue chiSquareDistribution
END
HISTOGRAM binsize 0.5 chiSquareDistribution 'Show the chi-square distribution
COUNT chiSquareDistribution >= observedChiSquare successCount
LET probability = successCount / numberOfTrials
PRINT probability
OUTPUT "Null hypothesis is "
IF probability >= 0.05
   OUTPUT "NOT "
END
OUTPUT "rejected at the 5% significance level.\n"
```

Tip: There is a set of subroutines in the "/lib" directory that will perform a chi-square analysis by resampling for problems involving contingency tables of any size. They are in the file *ChiSquareGeneral.txt*.

More Examples

You can find many other examples using *Resampling Stats* and *Statistics101* to solve a wide variety of common and uncommon problems in the folder "ResamplingPrograms" that is included with the *Statistics101* program by the installer. Further examples can be found at http://www.statistics101.net/PeterBruce_05-illus.pdf. Julian Simon's online text book at http://www.resample.com/intro-text-online/ gives a lengthy discussion of hypothesis testing and confidence intervals from the viewpoint of resampling. See especially, chapters 15 through 21.

The following examples are on the Statistics101 website. Click on any one of them to open that example in your browser.

Probability Examples:

1. Flipping three coins
2. Drawing two aces at random
3. One spade or one club
4. At least one head in two coin flips
5. Drawing either a spade or an ace from a deck of cards
6. Exactly five heads out of ten
7. Mean and standard deviation for 10 flips of a fair coin
8. Standard error of the mean
9. Area under normal curve
10. Percentile
11. Sixty boys out of next 100 births

Statistics Examples:

12. Confidence interval (SD known)
13. Confidence interval (SD known)
14. Hypothesis test 1 (SD known)
15. Hypothesis test 2 (SD known)
16. Confidence interval (SD known)
17. Hypothesis test (SD unknown. t distribution one tail)
18. Hypothesis test (SD unknown. t distribution two tail)

PART 2: Language Basics

In this section you will learn the fundamental concepts and commands of the extended *Resampling Stats* language. You will use these commands to create your own simulation programs. You may find that you need commands and features that are not described in this document. Since this is an introductory tutorial it does not cover all the capabilities of *Resampling Stats* and *Statistics101*. Therefore if you need something not described here, browse Appendix 2 or *Statistics101* program's help. It is very likely that what you need is already in the language as either a command or a subroutine. If you can't find what you want, feel free to post a question on the www.Statistics101.net web forum or contact me directly by email at john@statistics101.net.

Data Types

In the *Resampling Stats* language most data is in the form of lists of numbers. These lists are also called *vectors* or sometimes, *arrays*. (I will use the words *list* and *vector* interchangeably, with preference for the word *vector*). Each number in a vector is called an *element* of the vector. If a vector has a name it is called a *variable*. If a vector does not have a name, it is called a *constant*. If a vector has only one element, the vector may be called a *number*. Actually, the word *number* is used ambiguously in *Resampling Stats*. I've already used it two different ways in the preceding sentences. Sometimes it means a vector with only one element; sometimes it refers to the first element of a vector with multiple elements, and sometimes it just means a number in the sense that you are used to using it. The context should make clear which meaning is intended.

Although data can always be represented as numbers, often it is useful to use names instead of numbers as elements in the data. Named Constants are discussed later in this tutorial.

Numbers

Here are the rules for entering literal numbers in *Resampling Stats*. If a number you are entering is an integer, such as "5," you can just type it as "5" (without the quotes). If it is negative you type it as usual: "-5". It is never correct to use a plus sign with a number in *Resampling Stats*. For example, "+5" would cause an error message. If a number you are entering has a decimal point it MUST have at least one digit before the point, like this: "0.123" or "12.345". You can use scientific notation as in this example, "1.234E5", which stands for 123400.0. You can use a minus sign in the exponent and/or the mantissa, as in "-1.234E-5", but again, no plus signs.

Vectors (Lists)

Using vectors as the primary data type allows the handling of large collections of data as if they were a single entity. There are several different ways that you can write a vector in *Statistics101*. The most basic way of writing a vector is as a set of numbers separated by one or more spaces and enclosed between parentheses. A simple example of a vector is (1 2 3 4 5).

This next table shows the different ways you can represent vectors in *Resampling Stats*.

Specification Name	**Pattern**	**Example**	**Meaning of Example**
Empty vector	( )	( )	( )
Number	(n) or n	(3) or 3	(3)
Vector	(n1 n2 n3…)	(1 3 6 2.5 9)	(1.0 3.0 6.0 2.5 9.0)
Sequence	n1,n2	1,5 or	(1.0 2.0 3.0 4.0 5.0)
(two terms)		5,1	(1.0 2.0 3.0 4.0 5.0)
		jan,apr	(jan feb mar apr) See Named Constants
Sequence	n1,n2,n3	2,10,2	(2 4 6 8 10)
(three terms)		2,1,0.1	(2.0 1.9 1.8 1.7 1.6 1.5 1.4 1.3 1.2 1.1 1.0)
		jan,dec,3	Illegal: step not allowed in name sequences[1]
Multiple	n1#n2 or	5#3	(3.0 3.0 3.0 3.0 3.0)
	n1#vec	2#(1 2 3)	(1.0 2.0 3.0 1.0 2.0 3.0)
Multiple	n1#n2,n3,n4	2#3,5	(3.0 4.0 5.0 3.0 4.0 5.0)
Sequence		2#2,6,2	(2.0 4.0 6.0 2.0 4.0 6.0)

The first column, "Specification Name," gives the name of each way of creating a vector. The "Pattern" column, where "n," "n1," "n2," etc. represent numbers and "vec" represents a vector, shows different generic ways to write a vector in *Resampling Stats*. The "Example" column gives specific numerical examples of each pattern. The "Meaning of Example" column shows the full vector that is represented by the example.

The first row of the table shows an empty vector. Note that an empty vector is not the same as a vector whose sole element is zero. An empty vector has no elements.

The second row shows a "number," which is a vector with only one element. A number can be written with or without parentheses. This is a special case for convenience and for conformity to usual practice of writing numbers. *Statistics101* treats either notation as equivalent.

The third row shows a vector with several numbers in it. You use this when there is no particular pattern in the numbers that allows you to use one of the following more compact specification methods.

The next four rows in the table show how you generate certain kinds of vectors without having to type in all the numbers. Using the two term Sequence specification you enter only the lowest and highest values of a desired sequence, separating those two values with a comma. The resulting sequence vector will have all its elements differing from their adjacent element by 1. If the number at the left of the comma is lower than that on the right, then the sequence will be in order of increasing values. If the number at the left of the comma is higher than that on the right, then the sequence will be in order of decreasing values. The next table row shows that an optional third term may be added following a second comma to assign a "step" value that specifies how much each element differs from its preceding element. The step value must be positive, even when the values of the sequence are in decreasing order. Any or all of the two or three numbers in the Sequence spec may be integer or floating point literals, or variables.

[1] You can get the intended result like this: `TAKE jan,dec 1,12,3 result`

For the Multiple specification, you enter the number of times (n1) that you want another value (n2) or vector (vec) repeated, both separated by a pound sign (#), called the multiple operator. In the Multiple Sequence specification, you specify a sequence on the right side of the pound sign using any of the forms just described. Either or both sides of the multiple operator may be variables.

A *missing number* is an element of a vector for which you have no data, but you want to represent the fact that that element is not available. There are two special codes that can be used for missing numbers. The two codes are “NaN” and period (“.”). NaN is an acronym for “Not a Number.” The two codes are considered equivalent by *Statistics101*. For example, if you had a set of data of students' height versus weight, but for some students you had only their height or only their weight, you would mark the missing data with NaN or period like this:

```
COPY ( 62  71   60    68   NaN    72) heightData
COPY (102 115    .   170   145   194) weightData
```

Arrays

An array is an ordered collection of vectors that are referenced by a name and a set of one or more index numbers separated by enclosing square brackets. An example of an vector identifier that is a member of an array would be “myArray[n][3]”. The name, “myArray,” is called the array name, and the variable “n” and the number “3” are called indexes. The number of indexes that an array has is called its dimensions. The array myArray in this example is a two-dimensional array. An array can have any number of dimensions.

Arrays are created using the NEWARRAY command. Each NEWARRAY command creates one array and fixes its dimensions and their sizes. A NEWARRAY command creating the myArray array might be as follows:

```
NEWARRAY myArray 5 6
```

That command specifies that the array has two dimensions, the first having the range 1 through 5, and the second having the range 1 through 6. Altogether, the array has space for 5x6 = 30 vectors. Note that for the NEWARRAY command itself, the dimension limits are not enclosed in square brackets.

An array vector identifier can be used anywhere that a normal vector variable is legal. An array vector identifier must always include the same number of dimensions that were specified in its NEWARRAY command. For more information, go to the NEWARRAY entry in the *Statistics101* Help documentation.

Names

Later, you’ll see how commands, such as COPY, give a name to a vector. A named vector is called a variable. For now, I’ll just state the rules for a valid name: it must begin with a letter, followed by any number of letters or digits or underlines (_) or dollar sign ($) in any combination. Upper and lower case letters are considered to be the same.

The hyphen (-) is allowed in names, but in the context of a LET command, it may be confused with the minus sign and cause error messages. It is best to avoid its use.

Here are some valid names:

```
myName
standardDeviation$
class_grades
classGrades
class_1_grades
class1grades
```

And here are some invalid names along with the reasons they are invalid:

`2class`	Names can't start with a number. This will be interpreted as a number followed by a name.
`Class grades`	Names can't have a space. This will be considered to be two names.
`boys&girls`	Names can have only letters and digits, hyphens, underlines (_) and dollar signs ($). Anything else is an error.

When creating their own variable names, some people prefer to use underscores to separate words, while others prefer to use capital letters. You see examples of both in the valid names above. In the programs to follow in this document, I will use capital letters instead of underscores. It is a good idea to choose the style you like and stick with it consistently in a program. Doing so makes it more readable, at least by you.

Strings

A string is a sequence of text characters treated as a unit. In Resampling Stats there are two types of strings. The first type is called a literal string. A literal string is text contained between two double quote marks. For example, "This is a literal string." The second string type is called a variable string. A variable string is a text string that has its own name. Variable strings are created with the STRING command, which will be discussed briefly later. Strings are used as arguments to some commands. For example, strings are used by graphing commands such as HISTOGRAM to set the graph's labels.

Commands

The command is the unit of computation of a *Resampling Stats* program. Each command operates on data and produces a result. Every command is usually confined to one line[2] with the command name being the first item on the line followed by some number of "arguments." *Arguments* are numbers, variable names, or keywords that the command interprets. A *keyword* is a special optional word that modifies the behavior of a command. Only a few commands take keywords.

Of the 100-plus commands that are in the extended *Resampling Stats* language used by *Statistics101*, 30 are used most frequently. Once you understand how to use those 30 commands you will find it easy to add the others to your repertoire as needed. The complete list of commands is given in Appendix 2 of this document. The commands are grouped by categories in Appendix 3.

[2]You can continue a command onto subsequent editor lines by using a "continuation character". See the section Continuation Lines

Tip: If you don’t know or remember the name of a command or subroutine, or you don’t know whether a command or subroutine that you need already exists, you can do a keyword search of the Command/Subroutine Index to find the answer. You access the Command/Subroutine index via the *Statistics101* Help>Command/Subroutine Index menu item or by pressing the F3 key.

The most frequently used commands are:

Command	**Description**
ADD	Arithmetically adds corresponding elements of its input vectors.
COPY	Concatenates arguments into a single vector. An alternate name is DATA.
CONST	Assigns a list of numbers to a series of names. The names can then be used in place of the numbers.
COUNT	Counts the number of elements that pass a test.
DIVIDE	Arithmetically divides corresponding elements of its input vectors.
END	Marks the end of an IF, REPEAT, WHILE, FOREACH, UNTIL, or NEWCMD structure.
ENUM	Creates a set of enumerators, i.e., named constants whose names have no associated numerical value.
FOREACH	Executes commands between FOREACH and END assigning each element value of the given vector one by one to a variable.
GLOBAL	Makes the names of listed constants and variables visible throughout an entire program.
HISTOGRAM	Graphs a histogram of one or more vectors.
IF	Allows execution of commands between IF and END, ELSEIF or ELSE if a test is passed.
LET	Uses mathematical formula notation to compute a value and assign it to a variable.
MEAN	Computes the mean of a vector.
MEDIAN	Computes the median of a vector.
MULTIPLY	Arithmetically multiplies corresponding elements of its input vectors.
NAME	Creates one or more Named Constants. Combines the functions of the CONST and ENUM command.
NEWCMD	Creates a “new command” or “subroutine”.
NORMAL	Generates a vector of random numbers from a normal distribution.
PERCENTILE	Computes specified percentiles from an input vector.
PRINT	Prints the contents of one or more vectors to the output window.
REPEAT	Executes commands between REPEAT and END a given number of times.
SAMPLE	Randomly selects a given number of elements from a vector, with replacement.
SCORE	Accumulates results of random trials in a scoring vector.
SHUFFLE	Randomly reorders the elements of a vector.
STDEV	Computes the standard deviation of a given vector.
SUBTRACT	Arithmetically subtracts corresponding elements of its input vectors.
SUM	Adds all the elements of its input vector and puts the sum in the result vector.
TAKE	Copies specific elements from its input vector into its result vector.
VARIANCE	Computes the variance of a vector.
WHILE	Executes commands between WHILE and END as long as a test passes.

Although this document will always show the commands in upper case, you can use lower case if you prefer. *Statistics101* does not distinguish between upper and lower case.

The typical command follows this pattern:

```
COMMANDNAME optionalKeyword input1 input2 … inputN result
```

This means that the command, COMMANDNAME may have an optional keyword, takes the space-separated arguments `input1` through `inputN`, does whatever it is supposed to do with them and puts the result in a single output variable, here called `result`. Any preexisting contents of the result vector are erased and replaced with the command's output. As an example, suppose you have two vectors whose elements you want to add to each other. The ADD command can do the work for you (as you'll see later, you can also use the LET command to do addition as well as much more):

```
ADD vector1 vector2 vecSum
```

After ADD finishes, each element in `vector1` will have been added to the corresponding element in `vector2` and placed in the corresponding position in `vecSum`. If the two input vectors are of differing lengths, then the last element of the shorter one will repeated as many times as needed to make the two vectors' lengths equal, then they will be added. The shorter vector is actually unchanged; it is "virtually" extended only during the command's execution. Here is how ADD would work in a specific case:

```
vector1: (1 2 3 4 5 6)
vector2: (5 6 7)           (the 7 is "virtually" repeated 3 more times)
vecSum:  (6 8 10 11 12 13)
```

Like ADD, all the commands that operate on an element-by-element basis will virtually extend shorter vectors so that all the input vectors are the same length.

There is one atypical command, SCORE, which has the form:

```
SCORE input result1 result2 … resultN
```

This command takes only one input vector and can have one or more result vectors. The SCORE command, unlike all the others, does not replace the result vectors' contents. It appends the first element of vector `input` to the end of `result1`, the second element of `input` to the end of `result2`, etc. Its purpose is to remember the results of all the simulations that a program is performing. Normally you will use it with only one input and one result vector.

Command Syntax Descriptions

Since each command has its own number and type of arguments, it is necessary to have a simple and concise method of describing these features. The *Statistics101* program's help bar displays the syntax for each command using the following conventions:

An argument is described by its name. Keywords are shown capitalized. If an argument is enclosed in square brackets, [someArg], that argument is optional. If an argument is enclosed in curly brackets, {someArg}, that argument may be used zero or more times. If the argument name has neither square nor curly brackets, that argument is required. Ordinary parentheses are used to group required arguments. If a vertical bar, |, separates arguments or keywords, that indicates that a choice must be made between those arguments; only one is allowed. If an argument is enclosed between quotes, "someArg," that argument is a literal string. Here are some schematic examples of command syntax using these conventions and their explanations. CMD represents any command and A, B, C and D represent the arguments of the command.

`CMD A`	Command CMD requires one argument: A.
`CMD A B C`	CMD requires three arguments: A B and C.
`CMD A (B | C) D`	CMD requires three arguments: A B D or A C D.
`CMD A [B	C] D`	CMD requires two arguments (A and D) and optionally one of argument B or C following A.
`CMD A {B} C`	CMD requires A and C. Allows any number of Bs following A.

Taking a real example, the syntax description for the ADD command is:

```
ADD inputVector inputVector {inputVector} resultVariable
```

This means that the ADD command requires at least two input vectors and a result vector. Here's another example (this is all one line even though it had to be broken into three lines to print in this space):

```
HISTOGRAM [PERCENT] [BINSIZE binsizeNumber | BINS binsNumber] ["yAxisLabel" |
yAxisLabelStringVariable["xAxisLabel" | xAxisLabelStringVariabel]]
inputVector {inputVector}
```

This syntax description means that the HISTOGRAM command has an optional keyword, **percent**, a choice between either (or neither) of two keywords, **binsize** or **bins**, each of which requires an argument, an optional quote-enclosed string or string variable for the Y-axis label and an optional quote-enclosed string or string variable for the X-axis label, all followed by one or more input vectors. Also implied is that if there is an X-axis label there must be a preceding Y-axis label.

Comments

In *Resampling Stats*, anything to the right of an apostrophe (') is a comment intended for the human reader and is ignored by *Statistics101*. Comments are to help a reader, who might be you at some later date, to understand what a program is trying to accomplish. If your program has any complexity, it is a good idea to add comments to it for future aid in understanding it. The apostrophe may also be the first character on a line, in which case the entire line is a comment.

```
'This whole line is a comment
ADD A B C 'This is a partial line comment. This command adds A to B to get C.
```

There is a second kind of comment used in *Statistics101*, called the "block comment." The block comment may be contained entirely on one line or it may span multiple lines. Therefore, the block comment must be enclosed between two markers, a begin-comment marker (/*) and an end-comment marker (*/). If you are familiar with the languages C or Java, you will recognize this convention. *Statistics101* will ignore anything between the two markers. Only one asterisk is needed for either marker, but you may put as many as you like if you wish to highlight the comment block. Here is a short block comment:

```
/* Everything between the start
and end markers is a block
comment and will be ignored by Statistics101. */
```

Comments do not affect the execution speed of your program in *Statistics101*, so use them liberally. For further hints on how to improve your program's readability, see Program Clarity and Readability.

Continuation Lines

Normally you will type each command and all its arguments on a single line. A line in the editor can be as long as needed. But sometimes you might want to divide a command over more than one line. You can do this by ending each line of the command except its last with the continuation character, which is the backslash (“\”). For example, the command

```
HISTOGRAM percent binsize 0.1 "Probability Density" "X value" distribution
```

could in the extreme, be written over several lines like this:

```
HISTOGRAM percent \
  binsize 0.1 \
  "Probability Density" \  'Y axis label
  "X value" \              'X axis label
  distribution             'Last line has no continuation marker
```

Note that the last line must *not* have a continuation character. If you have a comment on a continuation line, the comment must follow the continuation marker as in the first two comments in the above example. If you put the continuation marker after the comment, it will be considered part of the comment and ignored, resulting in a syntax error message. Also, do not break a line in the middle of a variable name, keyword, or quoted string.

The LET Command

The purpose of the LET command is to make it easy to express complicated mathematical formulas. Here is an example using LET to compute the hypotenuse of a triangle:

```
LET hypotenuse = SQRT(sideA^2 + sideB^2)
```

Note that if sideA and sideB are vectors with many elements, then the above command computes the hypotenuse for each pair of elements, one element from sideA and one from sideB, and puts the result for each pair into the vector, `hypotenuse`. For example, the following program,

```
COPY (1 2 3 4) sideA
COPY (2 3 4 5) sideB
LET hypotenuse = SQRT(sideA^2 + sideB^2)
PRINT hypotenuse
```

produces this output:

```
hypotenuse: (2.23606797749979 3.605551275463989 5.0 6.4031242374328485)
```

Without the LET command, you would have to do something like the following to accomplish the same calculation and it would be much harder to comprehend at a glance:

```
SQUARE sideA sideAsquared
SQUARE sideB sideBsquared
ADD sideAsquared sideBSquared hypSquared
SQRT hypSquared hypotenuse
PRINT hypotenuse
```

Of course, you can use the arithmetic operators, +, -, *, /, %, and ^, which are respectively, add, subtract, multiply, divide, remainder, and power. There is also a vector concatenation operator, &, that joins the vector on its right to the vector on its left.

The minus sign “-” in its common usage has four different meanings: as a subtraction operator (a - b), as a “change sign” operator (-a), as an indicator that a literal number and/or its exponent is less than zero (-4.5E-3), and as a hyphen in a variable name ("class-grades"). When used following a variable name, you should put a space before the minus sign so it isn't interpreted as a hyphen. Otherwise, you will get a syntax error message.

You can also use literal vectors in the LET command. Rewriting the above calculation with literal vectors in the LET command (showing two ways to write literal vectors) gives this:

```
LET hypotenuse = SQRT((1 2 3 4)^2 + 2,5^2)
PRINT hypotenuse
```

As long as the result variable name (`hypotenuse` in this example) is not the same as that of a subroutine or a *Statistics101* command, you can omit the LET command name. *Statistics101* will detect the equal sign and interpret the line as a LET command. Rewriting the above without the LET keyword exemplifies this feature:

```
hypotenuse = SQRT((1 2 3 4)^2 + 2,5^2)
PRINT hypotenuse
```

You can even replace the COPY commands with LET commands like this:

```
sideA = (1 2 3 4)
sideB = (2 3 4 5)
hypotenuse = SQRT(sideA^2 + sideB^2)
PRINT hypotenuse
```

The LET command allows you to use all of the single-argument math command names, such as SIN and LOG, and some of the statistical command names, such as MEAN and MEDIAN as if they were functions, similarly to the way that SQRT is used above. In Appendix 3, all the commands that are legal to use with LET are shaded.

Furthermore, if you define a subroutine that has only one input variable and one output variable, you can invoke that subroutine within a LET command. For example, if you define a subroutine of the form

```
NEWCMD MYSUB inVec outVec
   . . .
END
```

where you can use any name you like instead of "mysub," you can invoke it in a LET command as you would any mathematical function:

```
. . .
LET resultVec = 3 * MYSUB(vecA) + COS(vecB)
. . .
```

For full information on the LET command see the *Statistics101* help documentation.

The STRING Command

The STRING command creates a single string variable by concatenating any number of literal strings, string variables, or vector variables. String variables can be used in any command that accepts a literal string argument. For example, you could use this command to create a string variable called `xAxisLabel`:

```
STRING "Sample Size = " size xAxisLabel
```

If `size` contained the number 30, then `xAxisLabel` would contain “`Sample Size = 30.0`”. The string variable `xAxisLabel` could then be used as the label for the X-axis of a histogram like this:

```
HISTOGRAM "Percent Frequency" xAxisLabel myData
```

The above command shows the use of a literal string (`"Percent Frequency"` as the Y axis label) and a string variable (`xAxisLabel` as the X axis label) in the same command at the same time.

If you want to be able to easily distinguish string names from vector names, you could adopt the convention of adding a "$" to the string name either as a prefix or a suffix as in “`$stringName`” or “`stringName$`.

There are two other commands that operate on strings, STRING_COMPARE and STRING_REPLACE. Look in the *Statistics101* help documentation for full details on all three string commands.

Programs

A program is an ordered sequence of commands designed to achieve some result. Since every program you write will need to display its output on the screen, let’s start with the PRINT command. Much of what you learn from the PRINT command will apply to the other commands as well. PRINT writes its arguments to the *Statistics101* output window, not to your printer.

Type the following into the empty *Statistics101* edit window:

```
PRINT 5 6 7
```

This one line is a very simple, but complete, program. The command name, PRINT, is first on the line, and is followed by some number of “arguments,” in this case the constants 5, 6, and 7. The arguments are separated by one or more spaces.

There are three ways to run this or any program in *Statistics101*:

- Click on the “Run” button ▶ in the *Statistics101* toolbar,
- Use the “Run Program” selection from the Run menu, or
- Use the keyboard shortcut, Ctrl-R.

Choose one of these ways and observe the result:

```
internal#001: 5.0
internal#002: 6.0
internal#003: 7.0
```

What does this mean? Our PRINT command had three numbers as its arguments. A number by itself, which is not enclosed in parentheses, is considered to be a vector containing only that number. So the PRINT command sees that it has three vectors, each containing one number, as if it had been written like this:

```
PRINT (5) (6) (7)
```

Since these three vectors had no names, *Statistics101* assigned names to them. These names differ only in their arbitrarily assigned numerical suffix. Then it printed each name one to a line, followed by a colon, followed by the value associated with that name. Note that the number-sign

(#) is not legal in any names that you can create. It is only used in these internally generated names.

Let's change our example slightly:

```
PRINT (5 6 7)
```

When you run this program using one of the three methods above, you see:

```
internal#001: (5.0 6.0 7.0)
```

This time, because you enclosed the three numbers in parentheses, thus telling *Statistics101* that the numbers constituted a vector, you got only one temporary variable and it was assigned to a vector made up of our three numbers. As an aside, although you entered the numbers as integers, they printed out as floating point numbers, i.e., they printed out as numbers with decimal points. *Statistics101* does all its arithmetic in floating point, and the output will normally be in floating point.

Normally, you won't see these internally assigned names because there is no need to print out constants. You assign names to important values in your program using almost any other command in the repertoire. The most common would be the COPY command. COPY, like most *Resampling Stats* commands, takes two or more arguments, the last of which must be a name that will be assigned to the vector that the command builds. You could experiment using the COPY command like this:

```
COPY 5 6 7 myData
PRINT myData
```

Here, COPY has 4 arguments: three numbers and a name for the result. When you run this two-line program using one of the three methods described earlier, you will see:

```
myData: (5.0 6.0 7.0)
```

What has happened? The COPY command automatically concatenates all its arguments except the last into a single vector and gives that vector the name of its final argument. So here, COPY put the three numbers into a vector named `myData`, and then the PRINT command printed the name, followed by the colon, followed by the list of numbers enclosed in parentheses.

I could have written:

```
COPY (5 6 7) myData
PRINT myData
```

This time, COPY has only two arguments: the vector (5 6 7) and a name for the result. In this second COPY command, although the data is entered differently, the result is exactly the same as before:

```
myData: (5.0 6.0 7.0)
```

The COPY command is used to create a vector filled with data from literal vectors and/or variables. Here are some examples of literal vectors used with COPY that show the result of printing out the created vector:

Command	Result
`COPY (1 2 3 4 5) A`	`A: (1.0 2.0 3.0 4.0 5.0)`
`COPY 1,5 B`	`B: (1.0 2.0 3.0 4.0 5.0)`

Command	Result
`COPY 1 2 3 4 5 C`	`C: (1.0 2.0 3.0 4.0 5.0)`
`COPY 5#3 D`	`D: (3.0 3.0 3.0 3.0 3.0)`
`COPY 5,2 (2 4 6) 3#7 E`	`E: (5.0 4.0 3.0 2.0 2.0 4.0 6.0 7.0 7.0 7.0)`
`COPY 2#3,5 F`	`F: (3.0 4.0 5.0 3.0 4.0 5.0)`

You can also use variables as input arguments to COPY. Here is a simple program solely to illustrate the use of variables in COPY:

```
COPY 1,5 A   'Create vector A containing the numbers 1 through 5
COPY 5#3 B   'Create vector B containing five threes
COPY A B C   'Concatenate A and B putting the result in C
PRINT C      'Print C
```

If you didn't need the vectors A and B for any other purpose in your program, you could dispense with them, replacing the above program with these two lines:

```
COPY 1,5 5#3 C
PRINT C
```

Both programs produce this result:

```
C: (1.0 2.0 3.0 4.0 5.0 3.0 3.0 3.0 3.0 3.0)
```

Occasionally you might find that you need to concatenate one or more vectors into one of those same vectors. For example, say you had two vectors, A and B, and you wanted to put the contents of A onto the end of B. Here is a command that will do so:

```
COPY B A B
```

This command copies the existing contents of B followed by those of A into B, replacing B's original contents. If you wanted to do it in the reverse order, i.e., you wanted to put A onto the front of B, you would write:

```
COPY A B B
```

This works for any number of vectors. If you had three vectors, A, B, and C, and wanted to put A then B then C into B, this command will do it:

```
COPY A B C B
```

The COPY command has a number of synonyms or aliases. The synonyms are CONCAT, DATA, NUMBERS, and URN. In the original *Resampling Stats* language, these all had somewhat different behaviors. For *Statistics101*, the features of all those commands were consolidated into one command and that command was allowed to be invoked by any of the listed names. You can always use the name COPY, or if you want to emphasize the purpose of the copy you can choose one of the other names. For entering data, for example, you might choose to use the name DATA instead of COPY.

Now, let's turn to a simple example that might have some practical application. Say you have the test scores from a class of 15 students and you want to know the average grade and the standard deviation. Here is how that might look as a *Resampling Stats* program:

```
DATA (89 98 69 83 58 40 78 93 85 60 72 83 79 83 76) scores
STDEV pop scores standardDeviation
MEAN scores average
```

```
PRINT average standardDeviation
```

The STDEV command computes the standard deviation of the `scores` vector and puts the result in the variable called `standardDeviation`. STDEV can compute the standard deviation of either a sample or a population. In this example the vector `scores` contains the entire population, not just a sample. The keyword **pop** tells the command to use the population formula for standard deviation instead of the sample formula. The MEAN command computes the mean of the elements of the scores vector and puts it in the variable called `average`. PRINT then produces the following output:

```
average: 76.4
standardDeviation: 14.42590262918292
```

Notice that I have given meaningful names to the variables/vectors in the program—no more "A" or "B" as names. The meaningful names come from the context of the problem the program is solving. I use A and B as names in this document when the programs are just illustrating a pattern and not solving a meaningful problem. If you use meaningful names for your variables, your program becomes much more understandable to anyone reading it.

As suggested earlier, some commands take *keywords* that modify the command's behavior. PRINT is another example of such a command. If the keyword **table** immediately follows the PRINT, then the PRINT command will print its arguments as a table. Each argument will be printed as a column; the first argument will be the first column, the second argument will be the second column, and so on. The table headings will be the names of the vectors. For example, the program

```
'Print a table of squares and square roots.
COPY 1,10 Number
MULTIPLY Number Number Square
SQRT Number SquareRoot
PRINT table Number Square SquareRoot
```

produces the following table as its output.

```
Number            Square            SquareRoot
1                 1                 1
2                 4                 1.414214
3                 9                 1.732051
4                 16                2
5                 25                2.236068
6                 36                2.44949
7                 49                2.645751
8                 64                2.828427
9                 81                3
10                100               3.162278
```

The PRINT command has other options you can learn about by reading its help text. Help is available in the *Statistics101* program via the menu selection Help>Help.

Logical Expressions

A *logical expression* is a comparison between values that produces a true or false result. Logical expressions are used to restrict the application of a command to only the elements that cause the result of the logical expression to be true. There are two types of logical expressions, *simple tests* (or just *tests*), and *compound tests*.

Simple Tests

An example of a simple test would be “`A<B`” which can be read as “is A less than B?” This test evaluates to true if A is less than B and to false otherwise. The “less than” sign, “<”, is called a *comparison operator*. There are six commands that require simple tests. These are COUNT, MULTIPLES, RECODE, RUNS, TAGS, and WEED. (The commands IF, ELSEIF, UNTIL, and WHILE can accept compound tests as well as simple tests. These commands will be described later.) A test restricts a command so that it applies only to the elements in its input vector that satisfy the test. The command uses the test to compare either the first element or each element of an input vector with one or two specified values or a list of values.

Let’s take the following simple program as an example:

```
COPY 1,5 5#3 C
COUNT C = 3 D
PRINT C D
```

I’ve used the command COUNT, which requires a test. COUNT counts all the elements of its input vector, “C,” that pass its test, “= 3” (the space between the test operator and the 3 is optional). So this COUNT command says “count how many elements of the vector C are equal to 3 and put the result in D.” When you run this program, you find:

```
C: (1.0 2.0 3.0 4.0 5.0 3.0 3.0 3.0 3.0 3.0)
D: 6.0
```

You will see other examples of the use of tests in programs throughout this book.

The complete list of test operators is described in the next table. Except for the memberof and notmemberof tests, if the right hand side argument of a test is a vector, only the first element of the vector is used in the test. If the left hand side (input) argument is a vector, then it depends on the command as to whether the test applies to only the first element (as in the IF command) or to all the elements (as in the COUNT command) of the input vector. As usual, the typographical case of the test keywords does not matter. Thus notMemberOf, notmemberof, NOTmemberof are all equivalent.

With regard to tests on missing data (NaN or “.”), NaN is equal only to NaN, not to any other number. Furthermore, NaN is not greater than or less than any number, including NaN.

Comparison Operators

Operator	**Description**
>	Greater than. “The element on the left is greater than the number on the right.” E.g., the following command counts the number of elements of `Z` that are greater than 5 and puts the result in `result`. `COUNT Z > 5 result`
<	Less than. E.g., the following command counts the number of elements of `Z` that are less than 5 and puts the result in `result`. `COUNT Z < 5 result`

Comparison Operators

Operator	Description
=	Equal. E.g., the following command counts the number of elements of `Z` that are equal to 5 and puts the result in `result`: `COUNT Z = 5 result`
<>	Not equal. E.g., the following command counts the number of elements of `Z` that are not equal to 5 and puts the result in `result`. `COUNT Z <> 5 result`
>=	Greater than or equal. E.g., the following command counts the number of elements of `Z` that are greater than or equal to 5 and puts the result in `result`. `COUNT Z >= 5 result`
<=	Less than or equal. E.g., the following command counts the number of elements of `Z` that are less than or equal to 5 and puts the result in `result`. `COUNT Z <= 5 result`

memberof	The element on the left is a member of the vector on the right. E.g., The following command counts how many elements of `Z` are members of `vectorA` and puts the answer in `result`.
	`COPY (1 4 15 22 39) vectorA`
	`COUNT Z memberof vectorA result`

notmemberof	The element on the left is NOT a member of the vector on the right. E.g., The following command counts how many elements of `Z` are not members of `vectorA` and puts the answer in `result`.
	`COPY (1 4 15 22 39) vectorA`
	`COUNT Z notmemberof vectorA result`

between	The element on the left is between (inclusive) the two numbers on the right. E.g., The following command counts the number of elements of `Z` that are between 1 and 10 and puts the answer in `result`. Note: there is no comma between the 1 and the 10.
	`COUNT Z between 1 10 result`
	Also, the limits may be in reverse order and it still works:
	`COUNT Z between 10 1 result`

notbetween	The element on the left is NOT between (inclusive) the two numbers on the right. E.g., The following command counts the number of elements of `Z` that are not between 1 and 10 and puts the answer in `result`. Note: there is no comma between the 1 and the 10. `COUNT Z notbetween 1 10 result` Also, the limits may be in reverse order and it still works: COUNT Z notbetween 10 1 result

Compound Tests

Sometimes you need to express a compound test. A compound test is composed of one or more simple tests combined with a "NOT," or an "AND," or an "OR," or an "XOR" (exclusive-or) condition. As defined above, a "logical expression" is either a simple test, such as `a < b`, or a compound test. Thus, wherever a logical expression is allowed, such as in an IF, ELSEIF, UNTIL, or WHILE command (which I will discuss later), a simple test is also allowed. The reverse is not true.

For example, you might want to test whether a card hand had two clubs and three hearts. This is an "and" condition. Or, you might want to test whether the hand had two clubs or three hearts. This would be the "or" condition. Or, you might want to test whether the hand had two clubs or three hearts but not both. This is the "exclusive-or" case.

Here are examples of all three cases. The examples assume that the variables `clubCount` and `heartCount` already contain the number of clubs and hearts that were in some simulated card hand. First, the "and" case, expressing "two clubs AND three hearts":

```
clubCount = 2 AND heartCount = 3
```

The above line means that if the variable `clubCount` is two and the variable `heartCount` is three, then this logical expression evaluates to true. Otherwise it evaluates to false.

Next, for the "inclusive-or" case, expressing "two clubs OR three hearts OR both":

```
clubCount = 2 OR heartCount = 3
```

And, finally, the "exclusive-or" case, expressing "two clubs OR three hearts, BUT NOT BOTH":

```
clubCount = 2 XOR heartCount = 3
```

The above examples are fairly easy to write and to interpret even without knowing the details of the logical operators NOT, AND, OR, and XOR. To use them for more complicated logical expressions, you must understand their properties and default behaviors as described in the following table. The term "precedence" refers to the order in which an operator will be applied in cases where a decision must be made between two or more operators. Higher precedence operators will be applied before lower precedence operators. Operators of equal precedence will be applied in order from left to right.

Logical Operators

Operator	**Precedence**	**Description**
NOT	1 (highest)	Logical NOT. Takes only one test argument, the one on its right. Results in true if the test on its right is false. Results in false if the test on its right is true. Highest precedence of all the logical operators. Example: `IF NOT A = 4` which, by the way, happens to be equivalent to `IF A <> 4`

Logical Operators

Operator	Precedence	Description
AND	2 (middle)	Logical AND. Takes two test arguments, one on its left and one on its right. Results in true if both the test on its left and the test on its right are true. Results in false otherwise. Precedence is below that of NOT and above that of OR and XOR. Example: `IF A > 5 AND B between 1 2`
OR	3 (lowest)	Logical inclusive OR. Takes two test arguments, one on its left and one on its right. Results in true if either the test on its left is true, or the test on its right is true, or both are true. Results in false otherwise. XOR and OR are of equal precedence. Theirs is the lowest precedence. Example: `IF A < 5 OR B <> C`
XOR	3 (lowest)	Logical exclusive OR. Takes two test arguments, one on its left and one on its right. Results in true if either the test on its left is true or the test on its right is true, but not both. Results in false otherwise. XOR and OR are of equal precedence. Theirs is the lowest precedence. Example: `IF A = 5 XOR B = 15`

The simplest form of logical expression is just a test, say,

```
a <= b
```

One step up in complexity is this next logical expression, which happens to be equivalent to the above:

```
NOT a > b
```

You can read this as “a is NOT greater than b.”

In the following examples the terms “test1” and “test2” etc. are used as a “shorthand” to represent the arguments of the logical operators, which are tests of the form “a = b,” “a < b,” “a between b c,” or any other test listed in the test operator table. Thus, a complete logical expression such as

```
a = b AND c <> d OR e between 1 2 AND f < g
```

can be described like this in that shorthand:

```
test1 AND test2 OR test3 AND test4
```

This shorthand is just for the purposes of visually simplifying the following examples in this tutorial. It is not valid *Resampling Stats* syntax and cannot be used in the *Statistics101* program. I now use the shorthand to give specific examples that will clarify the application of the precedence rules. Any logical expression represented by,

```
test1 AND test2 OR test3 AND test4
```

is equivalent to

```
(test1 AND test2) OR (test3 AND test4)
```

because of the default precedence rules. AND operators will be executed prior to OR operators. Another example, this time adding a NOT:

```
test1 AND NOT test2 OR test3 AND test4
```

The above logical expression is equivalent to:

```
(test1 AND (NOT test2)) OR (test3 AND test4)
```

A good way to understand how the precedence rules apply in the absence of parentheses that override the defaults is this: First all NOTs bind to the test that follows them, second, all ANDs bind to their arguments, and third, all ORs bind to their arguments that are the results of the first two steps. In the second step, if there are several ANDs in sequence they are all grouped together as in this next comparison of equivalent expressions:

```
test1 AND test2 AND test3 OR test4 AND test5 AND test6
```

The above logical expression is equivalent to:

```
(test1 AND test2 AND test3) OR (test4 AND test5 AND test6)
```

Applying the rules to this next expression, which has several ORs in sequence with an AND in the middle,

```
test1 OR test2 OR test3 AND test4 OR test5 OR test6
```

results in this equivalent expressed using parentheses:

```
test1 OR test2 OR (test3 AND test4) OR test5 OR test6
```

If you have any uncertainty about how the defaults apply in a complex logical expression that you write, you should use parentheses to coerce the expression to have the meaning you want.

Compound Commands

So far, our programs have been a simple sequence of commands that executed one by one until the last one was completed. A sequence of commands is the basic building block of a computer program. The real power or the computer comes from its ability to “decide” whether to execute some series of instructions or to repeat a series of instructions many times with slight changes. The commands that perform these decisions and loops in *Resampling Stats* are IF, REPEAT, FOREACH, WHILE, UNTIL, and NEWCMD. Each of these commands controls a sequence, or “block,” of other commands. Therefore, they are called “compound commands.” In each case, the block of commands is terminated by an END command. The simplest of the six compound commands is the REPEAT command so I’ll start there.

REPEAT ... END

Enter the following into the *Statistics101* edit window:

```
REPEAT 5
   COPY 1,3 A
   PRINT A
END
```

The REPEAT command takes only one argument. In this case, the argument is 5. The argument tells it how many times it is to repeat the commands that are between REPEAT and its END command. The argument may be any number; it can even be a vector, but if it is a vector, only the first element is used, the rest are ignored.

Tip: Notice that the two commands between the REPEAT and the END are indented. That is done just to make it easier to identify the block of commands that are controlled by the REPEAT command. It is not required, but it is a good practice to indent such blocks. You can have the *Statistics101* program do the indenting for you automatically by choosing one of the "Indent Program" selections from the Edit menu or the Toolbar, or by pressing the "I" key while holding down the Ctrl key on your keyboard.

When you run the above program, you will get this result:

```
A: (1.0 2.0 3.0)
A: (1.0 2.0 3.0)
A: (1.0 2.0 3.0)
A: (1.0 2.0 3.0)
A: (1.0 2.0 3.0)
```

The program has executed the COPY and the PRINT commands five times. If you think carefully about the program, you'll realize that the COPY statement copies the same numbers into the variable A over and over. Since that only needs to be done once, it is more efficient to move the COPY command out of the loop like this:

```
COPY 1,3 A
REPEAT 5
  PRINT A
END
```

Now, the COPY executes once and the PRINT executes five times. The computer is therefore doing less busy-work. It is good to be alert for these kinds of optimizations in the programs you write.

Many times you will write a program that follows this pattern:

```
...
REPEAT 1000
   ...                          'compute result
END
LET probability = result / 1000 ' divide result by number of trials
PRINT probability
```

In this pattern, you divide the result by the number of trials to calculate the probability. If you want to change the number of trials, you need to change it in two places. A more flexible way is to use a variable as in the following:

```
...
COPY 1000 numberOfTrials
REPEAT numberOfTrials
   ...                               'Compute result
END
LET probability = result / numberOfTrials
PRINT probability
```

Now you need only to change the number of trials once, in the COPY command.

FOREACH ... END

Like the REPEAT command, the FOREACH command repeats all the commands that are between the FOREACH and its matching END command, but its rule of repetition is different. The FOREACH command takes two arguments. The first argument is a variable that takes on, one by one, the value of each of the elements of the second argument. Each time through the loop, the variable is assigned the next value from the second argument and the commands between FOREACH and END are executed using that new value.

A simple example will make this clearer. In math, the “factorial of N,” written as “N!”, is the product of all the integers between 1 and N inclusive. Here is a program to compute a table of factorials:

```
COPY 1,10 Numbers
FOREACH number numbers
   PRODUCT 1,number factorial 'Compute product of numbers from 1 to number
   SCORE factorial Factorials 'Save the result in vector, Factorials
END
PRINT table numbers factorials
```

And here is its result:

```
Numbers       Factorials
1             1
2             2
3             6
4             24
5             120
6             720
7             5040
8             40320
9             362880
10            3.6288E06
```

You can read the FOREACH statement as if it said “for each `number` in the vector `numbers`, do the following.” Each time through the loop, `number` is assigned the next value from the `numbers` vector, then the commands in the body of the loop (PRODUCT and SCORE) are executed. After the commands are executed for each value in the numbers vector, the results are printed out as a table. You can compare this program with the earlier one that computed a table of squares and square roots.

You can use a sequence specification as the vector argument of the FOREACH command, like this:

```
FOREACH n 20,40,2
. . .
END
```

This iterates through all the even numbers between 20 and 40. The first two parts of the sequence specification can be variables and/or named constants. The third part, if present can be a variable, a number, or a named value (but not an enum).

There are more examples using FOREACH in the section on Nesting Commands.

IF ... ELSEIF ... ELSE ... END

The IF command takes a test, or more generally, a logical expression, as its argument. IF has three forms. The simplest form executes the commands between IF and END only if the result of the test is “true.” Here’s the outline of the simple IF command:

```
IF logicalExpression
   ' Commands to be executed if logicalExpression is true
END
```

This means that the IF command takes a logicalExpression as an argument, then is followed by a list of commands, followed by an END command. As a simple example, look at this program segment:

```
...
IF A <10
   PRINT A
END
```

Assume that in some earlier part of the program A has been given a value. Then in this part, if the value of A is less than 10, the PRINT statement will execute and print out the value of A. If A was a vector with more than one element, only the first element is used to determine the result of the test. To clarify this point, type this program into the *Statistics101* edit window:

```
COPY (3 5 7 8 9 20 32 55) A
COPY (10 2 15 22 19) B
IF A<B
   PRINT A
END
```

If you run this program you will see this result in the output window:

```
A: (3.0 5.0 7.0 8.0 9.0 20.0 32.0 55.0)
```

The test passed and A was printed because the first element (3) of A was less than the first element (10) of B.

Returning momentarily to the examples from the section on logical expressions, I can now show how those expressions are used in IF commands. Each example uses an IF command and assumes that the variables `clubCount` and `heartCount` already contain the number of clubs and hearts that were in some simulated card hand. First, the “and” case, expressing “two clubs AND three hearts”:

```
IF clubCount = 2 AND heartCount = 3
   LET successCount = 1 + successCount
END
```

This command means that if the variable `clubCount` is two and the variable `heartCount` is three, then record this as a success by adding a 1 to the variable named `successCount`. Otherwise, skip to the command that follows the END command.

Next, for the “inclusive-or” case, expressing “two clubs OR three hearts OR both”:

```
IF clubCount = 2 OR heartCount = 3
   LET successCount = 1 + successCount
END
```

And, finally, the “exclusive-or” case, expressing “two clubs OR three hearts, BUT NOT BOTH”:

```
IF clubCount = 2 XOR heartCount = 3
      LET successCount = 1 + successCount
END
```

The second, more general form of the IF command includes an ELSE command. It looks like this:

```
IF logicalExpression
   ' Commands to be executed if logicalExpression is true
ELSE
   ' Commands to execute if logicalExpression is false.
END
```

Here is a very simple concrete example:

```
IF A < B
   PRINT A
ELSE
   PRINT B
END
```

In this form, if the test is true, the commands between IF and ELSE will be executed. If the test is false, then the commands between ELSE and END will be executed. So here, if the first element of A is less than the first element of B, then the program will print A; otherwise, it will print B.

The third and most general form of the IF command adds the ELSEIF command. This form looks like this:

```
IF logicalExpression1
   ' Commands to execute if logicalExpression1 is true
ELSEIF logicalExpression2
   ' Commands to execute if logicalExpression1 is false
   ' but logicalExpression2 is true
ELSEIF logicalExpression3
   ' Commands to execute if logicalExpression1 and
   ' logicalExpression2 are false but logicalExpression3 is true
ELSEIF logicalExpression4
   . . .
ELSE
   ' Commands to execute if all the above logicalExpressions failed
END
```

In this form, there can be any number of ELSEIF commands. Each ELSEIF command must have a logicalExpression as its argument. Only one set of commands will be executed even if more than one test would pass. The first test that passes is the one that will execute, and then the program will skip all the remaining ELSEIF and ELSE commands and continue after the END statement.

The ELSE clause is optional, but there can be no more than one ELSE. The ELSE command must not have an argument. The commands in the ELSE clause will only be executed if all the other tests in the IF...ELSEIF...ELSE...END block fail.

This most general IF block command can act as what is called in other computer languages a “case statement.” Suppose you had a simulation in which you had several kinds of animals, chosen at random and you wanted to do different things depending on which animal was selected. Here is how that might be done.

```
ENUM cat dog cow bird
COPY cat,bird animals  'copy the sequence from cat to bird into animals
'. . .
SAMPLE 1 animals animal 'choose one kind of animal at random
IF animal = cat
   'do cat commands
   PRINT "Meeoow!"
ELSEIF animal = dog
   'do dog commands
   PRINT "Arf arf!"
ELSEIF animal = cow
   'do cow commands
   PRINT "Moooo!"
ELSEIF animal = bird
   'do bird commands
   PRINT "Tweet tweet!"
END
'. . .
```

This example also introduces the ENUM command, short for "Enumerator." This command allows you to use names such as cat, dog, ace, deuce, red, blue, etc. that have meaning in your program but don't have to have numerical values. And, as you can see from the example, once you have created the names, they can be used to specify a sequence. See the section Program Clarity and Readability for complete information on the ENUM, CONST and NAME commands.

WHILE ... END

The WHILE command is similar to the IF command in that it takes a logical expression as its only argument. Like IF, WHILE also executes the commands between itself and the END command if the logical expression's result is true. The difference is that after executing its block of commands, WHILE performs the logical expression again and if it evaluates to true again, it will execute its command block again. This will continue until the logical expression evaluates to false. Thus, some part of the logical expression must change or the loop will never end. If the logical expression evaluates to false the very first time the loop is entered, the loop will be skipped. Here's an example:

```
COPY 0 A           'Give A the starting value zero
WHILE A < 3        'As long as A is less than three, do the following
   PRINT A         '   Print A
   LET A = 1 + A   '   Add 1 to A and put the result in A
END
```

In this example, A changes its value each time through the loop, increasing by 1 each time. It is the command `ADD A 1 A` that causes A to be given a new value that is its old value plus 1. Eventually A will no longer be less than three and the WHILE loop will terminate. Here is the output of the program:

```
A: 0.0
A: 1.0
A: 2.0
```

The logical expression for the WHILE command can be as complex as you like. See the section Compound Tests for more information on logical expressions.

UNTIL ... END

The UNTIL command offers another looping option. Like the WHILE command, it requires a logical expression as its argument. This loop executes the commands between UNTIL and END until its logical expression becomes true. The UNTIL command tests its logical expression each time through the loop, but only *after* it executes its block of commands. That means that no matter what the logical expression's value is when the loop is first entered, the loop will execute at least once.

A *conditional loop* is a loop whose number of repetitions is not fixed but is determined by a logical expression. WHILE and UNTIL loops are conditional loops. REPEAT and FOREACH are unconditional loops. If you are writing a conditional loop that must execute zero or more times, then you should use a WHILE loop. If you are writing a loop that must execute one or more times, then you should use an UNTIL loop.

Since UNTIL depends on a logical expression, some part of the logical expression must change during the execution of the loop to cause the logical expression to become true at some point. Otherwise, the loop will never terminate.

Here is an example of a simple UNTIL loop:

```
COPY 0 A           'Give A the starting value zero
UNTIL A = 3        'Repeat until A becomes 3
   PRINT A         '   print A
   LET A = 1 + A   '   Add 1 to A and put the result in A
END
```

In this example, A changes its value on each cycle of the loop, increasing by 1 each time. The command `ADD A 1 A` causes A to be given a new value that is its old value plus 1. Eventually A equals three and the UNTIL loop will terminate. Here is the output of the program:

```
A: 0.0
A: 1.0
A: 2.0
```

You might be surprised to find that the following example is an infinite loop:

```
COPY 0 A
UNTIL A = 0
   PRINT A
   LET A = 1 + A
END
```

In this program, the variable A is set to zero, then the loop is entered. The loop doesn't evaluate its test until after it executes its first cycle, so it prints the current value of A, which is zero, then it increments A making it 1. Now, when the loop proceeds to evaluate the logical expression test it finds that A is not zero, so it goes through the loop again. The value of A will never be zero, so the loop will never terminate. WARNING: If you run this program in *Statistics101,* the loop is so "tight" that you might not be able to abort the program. If that happens, you will probably have to kill the *Statistics101* program using the Task Manager (Windows) or its equivalent in the Mac or Unix. Instead of running the program at full speed you can use the debugger to step through it.

Nesting Commands

The five commands, REPEAT, FOREACH, IF, UNTIL, and WHILE, can contain each other. This is called “nesting,” because one compound command is “nested” within another. Each END command belongs to the closest previous un-END-ed compound command. To make the concept concrete, here’s the skeleton of a program that has several nested commands.

```
...
REPEAT 1000
   ...
   IF A <= B
      ...
      WHILE C > D
         ...
      END
      ...
   END
   ...
END
...
```

The WHILE command and its included commands (represented by the “...”) are nested, along with other commands, within the IF command, which itself is nested within the REPEAT command. Almost every program example that I will discuss in this document or that you will write on your own will make use of nested compound commands.

Here is a simple real example using nested FOREACH loops to print out the *permutations* possible for rolling three die.

```
'List all the possible results (permutations) for rolling 3 die.
'For example, these six cases would be considered different and
'therefore each one would be printed:
'(1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1)
COPY 1,6 die
FOREACH face1 die
   FOREACH face2 die
      FOREACH face3 die
         COPY face1 face2 face3 roll
         PRINT roll
      END
   END
END
```

The first few results follow:

```
roll: (1.0 1.0 1.0)
roll: (1.0 1.0 2.0)
roll: (1.0 1.0 3.0)
roll: (1.0 1.0 4.0)
roll: (1.0 1.0 5.0)
roll: (1.0 1.0 6.0)
roll: (1.0 2.0 1.0)
. . .
```

Extending this example to show a more complex case, here is a way to show all the *combinations* that three dice can produce:

```
'List all the possible results of rolling 3 die without regard to
```

```
'their order (combinations). For example, these six cases would
'be considered equal and therefore only one would be printed to
'represent them all: (1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1)
FOREACH face1 1,6
   FOREACH face2 face1,6
      FOREACH face3 face2,6
         COPY face1 face2 face3 roll
         PRINT roll
      END
   END
END
```

Study carefully the differences between the two programs to understand how this second one works. Here are the first few results for comparison:

```
roll: (1.0 1.0 1.0)
roll: (1.0 1.0 2.0)
roll: (1.0 1.0 3.0)
roll: (1.0 1.0 4.0)
roll: (1.0 1.0 5.0)
roll: (1.0 1.0 6.0)
roll: (1.0 2.0 2.0)  '<-- First difference between outputs
. . .
```

The BREAK Command

The BREAK command forces immediate exit from the innermost REPEAT, FOREACH, UNTIL, or WHILE loop enclosing the BREAK command. Control will jump to the command following the innermost loop’s END command. Any loop enclosing the innermost loop will continue as if the innermost loop had completed normally.

For example, suppose you wanted to know on average how many random trials it would take to choose a given number, say 5, with replacement, from a list of all the integers between 1 and 10. The process to simulate is to create a list of the 10 numbers, select one at random with replacement. If that is not the desired number, select again, keeping track of how many trials it takes you to get the number. Here is a program to do that, using the BREAK command to exit the otherwise infinite WHILE loop.

```
DATA 1,10 numbers
COPY 5 desiredNumber
REPEAT 10000
   COPY 0 counter
   WHILE 1 = 1
      LET counter = 1 + counter
      SAMPLE 1 numbers theSample
      IF theSample = desiredNumber
         SCORE counter trials
         BREAK
      END
   END
END
MEAN trials meanCounts
MEDIAN trials medianCounts
PRINT meanCounts medianCounts
HISTOGRAM percent binsize 1 trials
```

The `WHILE 1 = 1` command is an endless loop because 1 is always equal to 1. The only way out is to use the BREAK command to force exit when the desired condition is met. When the BREAK executes, the WHILE loop terminates, but the REPEAT loop does not. It continues just after the WHILE loop's END command and then restarts the WHILE loop for the next trial.

I used the infinite loop in the above example program just to demonstrate the BREAK command. Try the following exercises:

Revise the program, using a non-infinite WHILE command and without the BREAK command.

Revise the program using a non-infinite UNTIL command.

Revise the program, using either WHILE or UNTIL to select the sample without replacement and compare and explain the differences in the result.

You can see my solutions at http://www.statistics101.net/selectingNumbersFromAList.html.

Program Clarity and Readability

When you write a simulation program in the *Resampling Stats* language or any computer language, it is a good idea to write it so that if possible, it explains itself to the reader with very few comments. The simplest contributor to readability is to indent the contents of the block commands, REPEAT, FOREACH, WHILE, UNTIL, IF, and NEWCMD. This was described earlier. *Statistics101* will perform the indentation automatically if you click one of the indentation icons in the toolbar or press Ctrl-I, Ctrl-Shift-I, or Ctrl-Alt-I.

Another helpful readability practice is to choose meaningful names for the variables you use in your programs. Meaningless names like "A1" or "xyz" are not helpful. If you use such names, you will have to "decode" them in your mind when you are writing or reading the program. The examples I provide throughout this document use variable names that make clear the meaning of the variable's contents.

A third practice that helps readability is to avoid or minimize the use of arbitrary code numbers in your program. You do this by creating meaningful names, called Named Constants, that you use in place of numbers. They are called constants because once you have created them you can't change their values. There are two situations where you might like to use names instead of numbers. The first is the case where you would otherwise have to use an arbitrary code number such as 1 for "heads" and 2 for "tails." The second is the case where you have a meaningful number that you want to give a name to, such as giving this number, 3.141592654, the name "pi." A Named Constant created for the first case is called an "enum," short for "enumerator." Named Constants created for the second case are called "Named Values."

Named Constants

As there are two kinds of Named Constants, there are two commands that you use to create them. To eliminate code numbers from your program altogether and just use names as if the name itself were a constant in the program, you would use the ENUM command. The constants you define with the ENUM command are called "enumerators" or "enums." You can assign a name to a constant number with the CONST command. The constants you define with the CONST command are called "Named Values." There is a third command, NAME, which combines the function of both commands. The next few paragraphs will describe these three commands in detail.

The rules for forming the names of constants are the same as the rules for variable names, which were described earlier in section Names.

Enums

Consider a simulation that involves flipping coins. In the original *Resampling Stats* language, you would have used a command like this next one to create the "coin":

```
COPY 1 2 coin  ' 1 = heads, 2 = tails
```

This is okay except that you've introduced some codes (1=heads and 2=tails) whose meaning you need to explain with comments and to keep in mind as you write or read the program. Using enums, you don't need to create codes. You just use the enums as if they were numbers. A typical example using enums for a simulation that involves "flipping coins" might be:

```
ENUM heads tails
COPY heads tails coin
```

or,

```
ENUM heads tails
COPY (heads tails) coin
```

Either creates a "coin" vector that contains the values "heads" and "tails." Note that by using the ENUM command, `heads` and `tails` become actual constants in the program. They do not have a numerical value. Their name is their value. *Statistics101* will not allow your program to change them. If you then print `coin` using the next command,

```
PRINT coin
```

you will see the result:

```
coin: (heads tails)
```

You might use these in a program like this next one that computes the probability of throwing exactly two heads in three tosses of a coin. This is the same program I used in the section The Resampling Method, but revised to use enums:

```
ENUM heads tails
COPY heads tails coin
COPY 1000 numberOfTrials
REPEAT numberOfTrials
   SAMPLE 3 coin trial      ' flip coin 3 times
   COUNT trial = heads headcount
   SCORE headcount results
END
COUNT results = 2 successes
LET probability = successes / numberOfTrials
PRINT probability
```

Enums can be used almost anywhere in the *Resampling Stats* syntax that a literal number (such as 1 or 1.234) can be used. The exception is that enums, since they do not have a numerical value, cannot be used in arithmetic commands such as ADD, SUBTRACT, etc. Therefore, if you have

```
ENUM sun mon tue wed thu fri sat
ENUM jan feb mar apr may jun jul aug sep oct nov dec
```

Then the following are legal commands:

```
COPY mon,fri workdays
COPY (sat sun) weekend
COPY jan,dec year
COPY 30#sun monthOfSundays
```

But these will cause an error message:

```
COPY tue,jun jumble  'ERROR: types of tue and jun don't match
ADD mon tue daySum   'ERROR: can't do arithmetic operations on enums
```

The first line is an error because the lower limit (`tue`) and the upper limit (`jun`) were created by different ENUM commands. All the enums in the same ENUM command are considered to be of the same category or "type"[3]. Enums from different ENUM commands are forbidden to be used as just shown because such a sequence specification is logically meaningless. There can be no such sequence as "Tuesday through June."

The second line is an error because the enums have no numeric value and therefore cannot be added.

As you saw earlier, the PRINT command will print Named Constants out as names, not numbers. Therefore, given the above ENUM and COPY commands, this command:

```
PRINT workdays weekend year
```

will produce this output:

```
workdays: (mon tue wed thu fri)
weekend: (sat sun)
year: (jan feb mar apr may jun jul aug sep oct nov dec)
```

Consider another easily understood example problem. The problem is "What is the probability that from a shuffled poker deck the first five cards will contain two clubs and three hearts?" (Not "*the* two *of* clubs" and "*the* three *of* hearts.") Here is a solution:

```
ENUM clubs spades hearts diamonds
COPY 13#clubs 13#spades 13#hearts 13#diamonds deck
COPY 100000 numberOfTrials
COPY 0 successCount
REPEAT numberOfTrials
   SHUFFLE deck shuffledDeck
   TAKE shuffledDeck 1,5 hand
   COUNT hand = clubs clubCount
   COUNT hand = hearts heartCount
   IF clubcount = 2 AND heartCount = 3
      LET successCount = 1 + successCount
   END
END
LET probability = successCount / numberOfTrials
PRINT probability
```

This program is completely self-documenting. There are no arbitrary codes to remember and no need for comments. The statement of the problem and the program itself are sufficient.

[3] The ENUM, CONST, and NAME commands have an optional keyword, **type**, that allows you to name the type created by the command. Multiple commands with the same type name behave as if they were created by only one command. You can find more information about this option in the Help documentation for the NAME command.

Named Values

If you want to assign your own code values to Named Constants, you can use the CONST command. You might want to do this where, as in the case of card hands, the cards might have special values and you want to use the names to represent the card value also. The next line exemplifies this case.

```
CONST (1 2 3 11 12 13) ace deuce trey jack queen king
```

This command assigns 1 to `ace`, 2 to `deuce`, etc. The first argument must be a literal vector and it must have one value for each of the subsequent names. Otherwise, you will get an error message. The first argument is allowed to be a sequence (e.g., 1,6) or a multiple specification (e.g., 6#1), although the latter would rarely be meaningful in the context of the CONST command since it gives the same value to all the names.

Of course, you can use the CONST command to give names to mathematical constants:

```
CONST (2.718281828 3.141592654) e pi
```

Numerical commands (such as ADD, LOG, SIN, etc.) operate on Named Values using their assigned values. If you had a card hand like the following,

```
COPY (ace jack king 4 6) hand
```

you could get its value like this:

```
SUM hand handTotal  'Sum all elements of hand and put result in handTotal
PRINT handTotal
```

The NAME command

The NAME command combines the functionality of both ENUM and CONST commands. The NAME command can create enums or Named Values, depending on the syntax you use. In early versions of *Statistics101* the NAME command was the only way to create Named Constants. The ENUM and CONST commands were introduced to emphasize the difference between the two types of constants but if you prefer, you can still use the NAME command. Here are examples of ENUM and CONST commands with their equivalent NAME commands:

```
ENUM mon tue wed thu fri sat sun  'These two commands are equivalent
NAME mon tue wed thu fri sat sun
```

and

```
CONST (1 2 3 11 12 13) ace deuce trey jack queen king
NAME  (1 2 3 11 12 13) ace deuce trey jack queen king 'These are equivalent
```

Most of the time, you will find the use of Named Constants intuitive. But there are cases where their behavior might not be what you expect. The next three subsections describe the details of the interactions between Named Constants and other features of the language so that you will understand the subtleties in case you run into an example that doesn’t match your intuition. You can skim or skip these on first reading, but remember that they are here and come back to read them if you run into any surprises using Named Constants.

Named Constants and Sequences

Named Constants can be used in sequence specifications. You have already seen examples of this earlier in this document. The rule is: If you specify a sequence of Named Constants, the sequence will be in the same order that you defined them in the command that created them. Any values you may have attached to them do not affect the sequence order. Here is an example that puts the weekdays in an odd order just to demonstrate the rule:

```
CONST (2 1 3 5 4) tue mon wed fri thu 'Odd order for demo purposes
COPY mon,fri weekdays
PRINT weekdays
```

Result:

```
weekdays: (mon wed fri)
```

As you can see, with a lower limit of `mon` and upper limit of `fri`, you get those limits plus whatever names are between them in the CONST command. The assigned values do not affect the order. If the limits are reversed, as shown here:

```
CONST (2 1 3 5 4) tue mon wed fri thu  'Odd order for demo purposes
COPY fri,mon weekdays
PRINT weekdays
```

You get this result:

```
weekdays: (fri wed mon)
```

Named Constants and Sorting Order

Since Named Constants can appear as elements of vectors along with normal numbers, the question arises as to how such vectors, called “mixed vectors” will sort. Here are the rules by which the sorting order is determined.

If your vector contains only enums (i.e., Named Constants to which you haven't explicitly assigned a value), the elements will be sorted into the same order in which they appeared in their respective ENUM or NAME commands. For example:

```
ENUM tue mon wed fri thu 'Odd order for demo purposes
COPY tue,thu weekdays
SORT weekdays weekdaysSorted
PRINT weekdaysSorted
```

Result:

```
weekdaysSorted: (tue mon wed fri thu)
```

If the vector contains only Named Values (i.e., Named Constants to which you have explicitly assigned a value), they will be sorted into order based on their assigned values, irrespective of their order in their CONST or NAME command. For example:

```
CONST (2 3 1 5 4) tue wed mon fri thu 'Odd order for demo purposes
SORT tue,thu weekdays
PRINT weekdays
```

Result:

```
weekdays: (mon tue wed thu fri)
```

If you SORT a mixed vector, which is a vector that contains both numbers and Named Constants, the Named Constants will fall into place based on the rules just discussed. Specifically, enums will come first (assuming ascending order) and Named Values and numbers will be positioned according to their values.

```
ENUM jan feb mar
ENUM mon tue wed
CONST (1 2 3) ace deuce trey
COPY mon,wed ace,trey jan,mar 1,3 mixedVector
SORT mixedVector sortedMixedVector
PRINT mixedVector sortedMixedVector
```

Which produces this result:

```
mixedVector: (mon tue wed ace deuce trey jan feb mar 1.0 2.0 3.0)
sortedMixedVector: (jan feb mar mon tue wed ace 1.0 deuce 2.0 trey 3.0)
```

Notice that the enums, having no assigned values, were grouped first with their siblings from the same ENUM command, and then sorted with the others of their group, and then the sorted groups were placed in the same order as their two NAME commands. Also, these enums were placed at the head of the vector.

The Named Constants that had assigned values are grouped with the numbers and then sorted according to their values.

Named Constants and Comparison Tests

Enums and Named Values behave differently in comparison tests. The difference is because enums do not have numeric values and therefore cannot be equal to or greater than or less than anything that does have a numeric value.

Enums compared with Enums, Named Values, and Numbers

An enum is equal only to itself. If an enum is compared for equality with a number or any other named constant than itself, the result will be false. An enum is less than another enum of its own type if it is earlier in its ENUM or NAME command than the other. An enum is greater than another of its own type if it is later in its ENUM or NAME command than the other. Any other comparison (such as a enum vs. a Named Value or vs. a Number or vs. an enum from a different ENUM or NAME command) yields false.

Named Values compared with Named Values and Numbers

A Named Value is equal to itself. If a Named Value is compared with another Named Value of the same type or with a number, then the result of the comparison is the result of the comparison of their values. If a Named Value is compared with a Named Value of a different type, they can never be equal even if they have the same assigned value. But any Named Value is less than any other Named Value that is created by a different CONST or NAME command later in the program.

The table on the next page shows some example comparisons that will help clarify the above rules, based on the following commands.

```
ENUM mon tue wed thu fri
ENUM jan feb mar apr may jun jul aug sep oct nov dec
CONST (1 2 3 10 11 12) ace deuce trey jack queen king
CONST (1 2) heads tails
```

Comparison	Result of Comparison	Reason

mon = mon mon < mon	true false	An enum is always equal to itself.
mon = jan mon < jan mon > jan mon <> jan	false false false true	mon and jan are different types because they are defined in different ENUM commands. Any comparison except "not equal" results in false.
mon = 1 mon < deuce mon <> deuce	false false true	An enum has no numerical value, so any comparison (except for "not equal") with a number or Named Value results in False.
ace < deuce trey > ace ace = trey	true true false	Named values of the same type (i.e., defined by the same CONST or NAME command) can be validly compared.
ace = heads ace <> heads ace < tails	false true false	Named values of different types (i.e., defined by different CONST or NAME commands) can only be "not equal."
ace = 1 heads < 2 heads <> 1	true true false	Named values can be validly compared with numerical values.

There are some predefined constants that you can use in your programs by INCLUDEing the files that define them. The files are in the lib directory:

The file **mathConstants.txt** defines the constants `pi`, `e`, `loge10`, `degToRad`, and `radToDeg`. The latter two are useful to convert from degrees to radians or the reverse by multiplication. To make all these math constants available to your program, just add the following command at the start of your program:

```
INCLUDE "lib/mathConstants.txt"
```

In earlier versions (prior to version 3.1) of *Statistics101*, there was a library file called **logicalConstants.txt** that defined the constants `true`, and `false`. To use the constants, the file had to be INCLUDEd at the beginning of a program. For versions 3.1 and newer of *Statistics101*, those constants are predefined automatically and no INCLUDE file is needed. They are defined as if they had been created by these commands:

```
NAME (0 1) false true
GLOBAL false true
```

That definition allows you refer to them as names or numbers in comparisons. Examples of their use can be found in the files lib/isSequenceCommand.txt and lib/chooseTest.txt.

If you want to access all constants defined in the library, you can INCLUDE just the file **constants.txt**, which in turn INCLUDEs all constant-defining files in the library. (As of version 3.1, it only includes the lib/mathConstants.txt file.) Here is a command that makes both math and any other constants available:

```
INCLUDE "lib/constants.txt"
```

Subroutines: User-defined Commands

A subroutine is a named set of commands that is treated as a unit. Using a subroutine can make a program shorter and easier to write and to understand. A subroutine is useful when

You have to perform the same computation in more than one place in a single program.

You have a computation that performs a generally useful function that can be used in many different programs.

You want to condense a complicated computation to one line to simplify the logic of a program.

There are three aspects of subroutines to understand: the *declaration*, the *definition*, and the *invocation.* The declaration introduces the subroutine’s name and argument list into a program, the definition provides the commands that the subroutine should execute, and the invocation calls for the execution of the subroutine at a certain point in your program.

Subroutine Declaration

A subroutine declaration introduces the subroutine’s name and argument list into your program. A declaration does nothing more than notify the *Statistics101* program that a subroutine by that name and with those arguments will be used later in the user’s program. There are two ways to declare a subroutine. The first, but least common, is to use the DECLARE command. DECLARE takes one or more arguments. Its first argument is the name of the subroutine. Any other subsequent arguments are arguments of the subroutine itself. DECLARE is only needed when for some reason you must invoke your subroutine in your program text before you have defined the subroutine. The second and by far most common way to declare a subroutine happens automatically as a byproduct of the NEWCMD command.

Subroutine Definition

In *Statistics101* a subroutine is declared and defined using the NEWCMD command. The NEWCMD command declares the subroutine and the rest of the commands up to the matching END command complete the subroutine’s definition.

NEWCMD takes one or more arguments. Its first argument becomes the name of the subroutine. Any other subsequent arguments become arguments of the subroutine itself. The subroutine's arguments are used to pass data into and/or out of the subroutine.

Normally, a subroutine's definition should precede its invocation in the program text. A subroutine may invoke one or more other subroutines as long as the invoked subroutines are declared prior to the invocation. If a subroutine is invoked before it is declared an error message will be forthcoming and the program will fail to execute. Therefore, it is good practice to put all the subroutines at the top of your program text, following any Named Constants and GLOBAL declarations.

Here's the definition of a simple subroutine that will copy the smaller of its two input arguments into its result argument:

```
NEWCMD MINIMUM number1 number2 result
   IF number1 < number2
      COPY number1 result
   ELSE
      COPY number2 result
   END
END
```

In this example, NEWCMD has four arguments. The first, `MINIMUM`, is taken to be the name of the subroutine. Its case doesn't matter, but since it will be used later as if it is a command, it is shown here in upper case. The rest of the arguments, `number1`, `number2`, and `result` indicate that subroutine `MINIMUM` requires three arguments. These latter arguments are called "dummy" or "formal" arguments because they only have meaning within the subroutine definition. They will be replaced with actual arguments, which may have different names, when the subroutine is invoked at run time. A subroutine may be declared to have a fixed or variable number of arguments, including none, and the arguments may pass values into and/or out of the subroutine. In this case, `number1` and `number2` pass data into the subroutine while `result` passes data out. The naming conventions for the subroutine's name and arguments are the usual *Statistics101* naming rules, and typographical case does not matter.

Visibility of Names

Dummy variable names are only valid within the subroutine in which they are declared. Any variable created inside a subroutine that is not a dummy variable is called a "local" variable. Any Named Constant that is created within a subroutine is called a local constant. (See the section Program Clarity and Readability of this document for more information on Named Constants) Local variables and constants are accessible only within the subroutine; they are not visible outside the subroutine that created them.

Any variable (or constant) created outside a subroutine is called a "non-local" variable (or constant). Non-local variables or constants may be referred to directly by name anywhere in the program after they are defined, except within a subroutine. The subroutine is like a black box. Nothing inside the box can see out and nothing outside can see in. With one exception, the only way in or out is via the subroutine's arguments. The exception is that variables or Named Constants declared outside a subroutine may be made visible within all subroutines by using the GLOBAL command. Non-local vectors listed as arguments to the GLOBAL command become "globals." Global constants and variables are accessible anywhere in the program, including inside subroutines, after the point of their declaration as globals in the text.

A dummy variable's name may be the same as that of a global or non-local vector, but it does not refer to that outside vector unless the global or non-local variable is put in the dummy's position during an invocation. A couple of examples will make this clearer. Suppose you have a program that includes some commands like this:

```
GLOBAL A
...
```

```
NEWCMD MYSUB B
   ...
   COPY A B    'Refers to global A
   ...
END
...
```

In the above subroutine, the A in the COPY command refers to the global variable, A. Now, if you have a program that includes the following structure,

```
GLOBAL A
...
NEWCMD MYSUB A   'Dummy A hides global A
   ...
   COPY A B      'Refers to dummy A
   ...
END
...
```

the A in the COPY command refers to the dummy variable, A. In other words, the dummy variable, A, hides the global variable, A.

Passing Data into and out of a Subroutine

Here is another simple example to illustrate several points. Suppose you wanted a subroutine to compute the log base 16 of a number. The formula is $\log_{16}(n) = \log_e(n)/\log_e(16)$. Here is one way to write the subroutine:

```
'Subroutine to compute the base 16 log of the elements of a vector
NEWCMD LOG16 input result
   LOG 16 logE16      ' log base e of 16 (a constant)
   LOG input logInput ' log base e of input
   DIVIDE logInput logE16 result
END
```

Within the subroutine, `logInput` is used to hold an intermediate result. Since `logInput` is not listed in the NEWCMD command as a dummy variable, it is a local variable. As such, it is not accessible outside the subroutine by name.

The first line of the subroutine (`LOG 16 logE16`) is computing a constant. Since it is inside the subroutine, it will be recomputed every time the subroutine is invoked, which would be inefficient if the subroutine will be called many times. There are four ways to avoid recomputing the constant. I will illustrate those four ways partly to demonstrate some of the different options you have when writing a subroutine and partly to show how the GLOBAL and NAME commands interact with subroutines.

One way is to compute the constant outside the subroutine and declare it global, like this:

```
LOG 16 logE16       ' log base e of 16 (a constant)
GLOBAL logE16

'Subroutine to compute log base 16
NEWCMD LOG16 input result
   LOG input logInput ' log base e of input
   DIVIDE logInput logE16 result
END
```

A second way is to use a Named Constant outside the subroutine, as in the following:

```
CONST 2.772588722239781 logE16  ' log base e of 16
GLOBAL logE16

'Subroutine to compute log base 16 of input
NEWCMD LOG16 input result
   LOG input logInput         ' log base e of input
   DIVIDE logInput logE16 result
END
```

Named Constants are created once, just before the program is run, so they have minimal “cost” at runtime. If the constant `logE16` is not needed outside the subroutine, it may be declared within the subroutine. This is the third way, as illustrated here:

```
'Subroutine to compute log base 16
NEWCMD LOG16 input result
   CONST 2.772588722239781 logE16  ' log base e of 16
   LOG input logInput              ' log base e of input
   DIVIDE logInput logE16 result
END
```

A fourth way to avoid recomputing a constant is to pass the constant into the subroutine as an argument:

```
'Subroutine to compute log base 16
'logE16 is the constant Log(16) base e, to be passed in.
NEWCMD LOG16 logE16 input result
   LOG input logInput ' log base e of input
   DIVIDE logInput logE16 result
END
```

The above examples illustrate four different ways of getting information into a subroutine. In summary, to pass a *variable* into a subroutine, the preferred way is to pass it as an argument. The non-preferred way is to declare it GLOBAL and refer to it directly inside the subroutine. To pass a *constant* into a subroutine, the preferred ways are either define it (using CONST or NAME) within the subroutine if possible, or define it (using CONST or NAME) outside the subroutine and declare it to be global (using the GLOBAL command).

Had I not been using the subroutine to illustrate the above points, I probably would have written it more straightforwardly in the following way:

```
'Subroutine to compute log base 16
NEWCMD LOG16 input result ?"Subroutine to compute log base 16"
   LET result = log(input) / 2.772588722239781
END
```

Optional Arguments

Many of the Statistics101 commands, such as COPY or ADD, take a variable number of arguments. Sometimes you might want to write a subroutine that is capable of accepting a variable number of arguments like one of those commands. If so, after any required dummy arguments on your NEWCMD command line, you would add a number-sign (#). If this code is present it indicates that the subroutine's invocation is allowed to have an unspecified number of additional, or “optional” arguments. If the # is not present, then the invocation must have exactly as many arguments as the NEWCMD command that defines the subroutine. The # mark may be followed by a double-quote-enclosed string which is a comment to describe the optional argument list. The comment will appear in the Syntax Help Bar near the top of *Statistics101*'s main window. If you omit the comment, the default comment “{variable}” will appear in the Syntax Help Bar.

Here is an example of a subroutine definition that calls for one required argument and some optional arguments.

```
NEWCMD EXAMPLESUB inVector #"{inVector} resultVector"
   . . .
END
```

The argument `inVector` is required when EXAMPLESUB is invoked because the argument is before the # sign. The # sign indicates to the *Statistics101* compiler that the subroutine EXAMPLESUB may be invoked with some unspecified number of optional arguments. The optional comment in quotes after the # is to describe what kind and how many optional arguments the subroutine expects. In this example, following the syntax conventions described in the section Command Syntax Descriptions, the comment tells the human reader that the subroutine expects zero or more input vector (`inVector`) arguments and one result vector argument (`resultVector`). The comment will be displayed in the syntax help bar but is otherwise ignored by the *Statistics101* compiler.

Inside a subroutine definition the *dummy* variables are referred to by their names. In the case of the *optional* variables (those allowed by the # mark), those are referred to by number using two other commands: ARGCOUNT, and GETARG. ARGCOUNT tells you how many optional arguments are in the invocation. GETARG returns one of the optional arguments, chosen by number. Here is an example of a subroutine showing the use of ARGCOUNT and GETARG.

```
'Subroutine to do an ascending coordinated sort, in place, of
'two or more vectors on the first vector (keyVariable) as the key.
'All vectors must be the same length.
NEWCMD SORTCOORD keyVariable #"variable {variable}" \
 ?"Coordinated sort, in place, of two or more vectors"
   ARGCOUNT numberOfArgs
   IF numberOfArgs > 0
      TAGSORT keyVariable tags
      TAKE keyVariable tags keyVariable
      FOREACH argNum 1,numberOfArgs
         GETARG argNum arg
         TAKE arg tags arg
      END
   ELSE
      DEBUG "ERROR: Incorrect number of arguments in SORTCOORD."
   END
END
```

Subroutine Invocation

The third aspect to writing subroutines is their invocation. A subroutine is invoked simply by using the subroutine's name as if it were a normal Statistics101/Resampling Stats command and listing the appropriate actual arguments in place of the dummy arguments. Here are two invocations of subroutine MINIMUM, which was defined in the earlier section, Subroutine Declaration:

```
MINIMUM 5 10 result
MINIMUM girls boys smallerGroupSize
```

The first example is simply to show that literals can be used as inputs. The second shows the use of variables for input. Notice that the names of the arguments in the invocation do not have to match those of the dummy variables in the declaration. The "actual arguments," `girls`, `boys`, and `smallerGroupSize` will take the place of the dummy variables, `number1`, `number2`, and `result`. The substitution is made strictly on the basis of position, not name. Thus, the first actual argument will be substituted for the first dummy argument, the second for the second, and so on.

Here is another example, showing the last of the LOG16 subroutines defined earlier along with a main program that invokes it. You can cut this program out, paste it into *Statistics101*'s edit window and run it.

```
'Subroutine to compute log base 16
NEWCMD LOG16 input result
   LOG input logInput             ' log base e of input
   DIVIDE logInput 2.772588722239781 result
END

'Main program using the subroutine
COPY 1,16 vec
LOG16 vec log16vec  'subroutine invocation
PRINT table vec log16vec
```

The main program creates a vector named `vec` containing the numbers from 1 through 16, then invokes the `LOG16` subroutine to compute the base 16 log of all of `vec`'s elements and put the results in a vector called log16vec. All the other above definitions of the LOG16 subroutine, except the next-to-last (repeated just below), can run with this same main program. The next-to-last one would work with a main program like the following, which defines the necessary constant to be passed in.

```
'Subroutine to compute log base 16
'logE16 is the constant Log(16) base e, to be passed in.
NEWCMD LOG16 logE16 input result
   LOG input logInput ' log base e of input
   DIVIDE logInput loge16 result
END

'Main program using the subroutine
COPY 1,15 vec
LOG 16 logE16              'compute constant log base e of 16 once
LOG16 logE16 vec log16vec  'subroutine invocation
PRINT vec log16vec
```

Subroutine Annotations

While it is creating the subroutine from the NEWCMD command, *Statistics101* will automatically collect the subroutine name and all its arguments for use as help text in the Syntax Help Bar. For example the earlier MINIMUM subroutine definition, repeated here for convenience:

```
NEWCMD MINIMUM number1 number2 result
   IF number1 < number2
      COPY number1 result
   ELSE
      COPY number2 result
   END
END
```

would cause the following to appear in the Syntax Help Bar when the cursor is on a line starting with the subroutine's name:

```
MINIMUM number1 number2 result
```

The help text serves as a reminder of the subroutine's arguments when you are writing a program using a subroutine.

In addition, there are two kinds of "annotations" that you can manually add to your subroutines to make them easier to use. These annotations further integrate your subroutines into the *Statistics101* help system. The first allows you to add a one-line description of the subroutine that will appear in the Description Help Bar just like the one-line descriptions for the built-in commands. The second allows you to specify one or more categories for the subroutine. These cause the subroutine's name to be included in the *Statistics101* category menus and in the category folders in the Subroutine Browser, which is described below in the section titled The Subroutine Browser.

Subroutine Description Annotation

NEWCMD accepts an optional one-line description that will appear in the Description Help Bar (just under the Syntax Help Bar) when the cursor is on a line starting with the subroutine's name. The subroutine's description is introduced by a question mark, "?" (as a symbol for "help"). The description must be enclosed within quotation marks. The description may be any length but must be all on one line. Here is the above subroutine revised to add a description (the description is in boldface):

```
NEWCMD MINIMUM number1 number2 result ?"Returns the minimum of two numbers"
   IF number1 < number2
      COPY number1 result
   ELSE
      COPY number2 result
   END
END
```

Note, though, that the help texts won't appear until after *Statistics101* has performed a syntax check. The syntax check is performed when you click on the syntax check button on the toolbar, or choose the Run>Check Program Syntax menu, or use the Control-K keyboard shortcut, or when you run the program.

Subroutine Category Annotation

You can place a subroutine into one or more categories using the "@" sign followed by a category name. If the category name, such as "financial," is a single word, it doesn't need to be enclosed in double quotes. If it is multiple words, such as "vector operations," then it must be enclosed in double quotes. Upper and lower case category names are equivalent.

The reason for assigning categories to a subroutine is so that the subroutine can be listed along with others of the same category in the "Commands and Subroutines By Category" menu list of the Edit menu, in the Edit Window popup menu, and in the Subroutine Browser. *Statistics101* builds the category lists based on the subroutines it finds in the *lib* directory. If you are not going to save your subroutine in *lib* or in your default subroutine folder (set via menu: Edit>Preferences...>General tab), then there's no need to use this feature.

If you assign a category that is not currently in the menus' lists, that category will be added to the menus and the Subroutine Browser. If you don't assign a category, the subroutine will be assigned the default category, "other." If you assign more than one category, the subroutine's name will appear in each category's list in the menus and the Subroutine Browser.

If you want a subroutine's name to not appear in the category menu, you can use the tag "@HIDE". If that tag is used, then all other tags on the same NEWCMD line are ignored. The subroutine's name will, however, still appear in the "Subroutines From Lib By Name" menu and in the Subroutine Browser.

Here is an example of a subroutine definition extended to assign the subroutine to two categories.

```
NEWCMD MINIMUMS inVec1 inVec2 result  @math @"vector operations" \
  ?"Copies minimum elements at each position into result."
   . . .
END
```

Note that the category annotations, if present, must precede the "?" description annotation. Remember that the "\" indicates that the command is continued to the next line.

Subroutine Libraries

You'll find a "library" of subroutines and constant declarations in the directory named *lib* in the *Statistics101* installation directory. Read the *Read_Me.html* file in the *lib* directory for a brief introduction to the available subroutines. These subroutines provide functionality that will save you time with many common problems.

All the subroutines that are in the *lib* directory will automatically be included in the "Subroutines From Lib By Name" and "Commands and Subroutines By Category" sub-menus in the menu bar's Edit menu and the edit window's popup menu. Those menus have two options:

- If you select a subroutine from either menu, the file containing it will appear in the help browser so you can read its descriptive comments and the text of the subroutine itself.
- If you hold down the SHIFT key while making a selection from either of those sub-menus, the name of the subroutine (or command) will be inserted in the Edit window at the cursor, and an INCLUDE command will be added to the top of the program if it is needed.

Statistics101 will also create a directory with the name "ResamplingStatsSubroutines" in your user home directory the first time that *Statistics101* runs. You can use that as a convenient place to store your own subroutines for easy access. Or, you can change the default to a directory of your choosing by opening the Edit>Preferences...>General tab and entering the directory's path in the "Default folder for my subroutines:" text box.

The subroutines that you store in your default subroutine directory will not be included in the popup menu but they will be scanned by *Statistics101* for inclusion in the Subroutine Browser and to collect the help annotations (those introduced by the "?" keyword) for use in the help bar.

Your subroutines will also be included in the Command/Subroutine Index (Help>Command/Subroutine Index or press the F3 key).

If you have some subroutines or even some constants that you use often, you can put them in one or more files in your default subroutine directory and then use the INCLUDE command to bring them into any other program that needs them.

The INCLUDE command

The INCLUDE command lets you use a subroutine in more than one program without having to copy it into each program. The INCLUDE command inserts files named in its argument list into the program text at the location of the INCLUDE command. The files named for inclusion should contain subroutines and/or constant definitions. The result is that a new expanded program is created temporarily by *Statistics101*, consisting of the original file with each of its INCLUDE statements replaced by the contents of all the files listed in its argument list. The full program with all the nested included files expanded can be viewed and debugged in the Debug tab of the Program Panel.

The INCLUDE command takes one or more file names as arguments. Each file name must be enclosed between double quotes. There must be at least one space between adjacent file names.

You can use the INCLUDE command to insert any subroutine file from the library into your own simulations, like this one that brings in some subroutines that manipulate the elements of vectors:

```
INCLUDE "lib/vectorCommands.txt"
```

Tip: If the auto-include feature is enabled (menu: Edit>Preferences...>Editor>Enable Auto-INCLUDE) and you invoke a subroutine that is defined in the lib or your default subroutine directory, then during syntax check *Statistics101* will automatically insert the appropriate INCLUDE command at the top of your program.

Tip: If you want to include a file but can't remember its name or don't want to type it, you can select the Edit>Include Lib File(s)... menu item (or press Ctrl-U). It will produce an Open File dialog that lists all the files in the lib directory. When you select one or more files and click the Open button, *Statistics101* will insert an INCLUDE command for each of your selected files at the top of your program.

Refer to the *Statistics101* help for more details on how to use the INCLUDE command.

The Subroutine Browser

All the subroutines that are included with *Statistics101* have descriptive comments just before each NEWCMD command explaining how to use the subroutine. You can read these comments to learn what each subroutine is for and how to use it, but to do that, you need to know what file the subroutine is in and then open that file and find the subroutine within it. The Subroutine Browser makes it easy to find any subroutine in lib and your default subroutine folder (defined in Edit>Preferences) along with its description and code.

There are two ways to access the Subroutine Browser. The first is via the Window>Show Subroutine Browser menu in the Statistics101 main window's menu bar. The second is via the popup menu that appears when you right-click in the *Statistics101* Edit window. If you select any subroutine name from that popup menu, the Subroutine Browser will appear and will display that subroutine's file scrolled to that subroutine's NEWCMD command.

Refer to the *Statistics101* help for more details on the Subroutine Browser.

Subroutine Summary

Here are some important points to remember about “new commands” or “subroutines”:

A subroutine is effectively a user-defined command. Once defined in a program, a subroutine’s name is used in the same program just like that of any other *Statistics101*/*Resampling Stats* command.

A subroutine cannot have the same name as any *Statistics101*/*Resampling Stats* command. Two subroutines in the same program cannot have the same name.

It is good practice to put all your subroutine definitions near the beginning of a program, after any NAMEd constant definitions and GLOBAL commands but ahead of the “main program.”

A subroutine may be declared anywhere in the program (except inside another subroutine's definition) as long as it is declared in the text before it is invoked. A subroutine is declared by a DECLARE command. A subroutine is defined (and declared) by a NEWCMD command. If you define the subroutines in your program text before their first invocation, you do not need to use the DECLARE command.

A subroutine may invoke other subroutines but only if the invoked subroutines are declared earlier in the program text. A subroutine may not invoke itself (recursion) and two subroutines may not invoke each other.

A subroutine must be invoked with exactly the same number of actual arguments as there are dummy arguments in its declaration. If the declaration has three dummy arguments, then every invocation of that subroutine must have three actual arguments. If the declaration allows optional arguments (by using the “#” keyword in the subroutine’s declaration), the optional arguments, if any, follow any required actual arguments.

A subroutine cannot directly access variables or constants that are declared outside itself. If it must access some variable or constant vector from the main program or an enclosing subroutine, then that vector must be passed to the subroutine in its argument list or that vector must be declared to be global with a GLOBAL command

All of a subroutine's local variables are cleared when the subroutine completes execution (i.e., reaches its END command). A subroutine cannot hold a value in a local variable between executions. If a subroutine needs to save a value from one execution to another, you can create a global variable to hold that value.

A subroutine’s name and all its arguments will be used as help text in the *Statistics101* main window’s syntax help bar. This help text will become effective after the first syntax check is performed on any program that includes a subroutine. If you are in the process of writing your program and the help text doesn’t appear when the cursor is on a line containing a subroutine invocation, then just run a syntax check by selecting menu Run>Check Program Syntax. Even if there are errors, the subroutine’s syntax help text will be established.

Error Messages

There are two kinds of errors you can make when writing a program: syntax errors and logic errors. *Statistics101* can detect all syntax errors, but it can only detect those logic errors that lead to absurdities such as attempts to perform an arithmetic operation on an enum (a named constant that was not assigned a numerical value), or trying some operation on an unintentionally null vector, or dividing by zero, or taking the log of a negative number, etc. If you make a logic error, most of the time you will have to detect it yourself, usually by judging whether the simulation results give a reasonable answer and carefully reviewing your program. The debugger can help you find logic errors.

When you run a program in *Statistics101*, say by clicking on the Toolbar's Run button, two things happen. First, *Statistics101* performs a syntax check of the program you've typed into the edit window. If the syntax is correct, then *Statistics101* goes ahead and executes your program. Errors can occur in either of these two phases, the syntax check phase or the execution phase.

When it detects an error, *Statistics101* will report the error in its output window and abort the program. If the error occurred during the execution phase, *Statistics101* will also open the debug window showing the state of all the program's variables just prior to the error.

If you double-click on an error message in the Output Panel, the view of your program in the Edit or Debug window will jump to the offending line, which will be highlighted in red.

Syntax Errors

Errors during the syntax check phase are frequently the result of misspelled commands, incorrect number of arguments, an incorrect character in an argument vector, inconsistently spelled variable names, or a named constant being used as a result variable.. The error message states *Statistics101*'s understanding of the problem and gives the line number on which the error was detected. Usually when you look at the designated line you will easily find the error. Occasionally (rarely), the designated line is incorrect and the actual error may be found a line or two prior to the designated line.

To avoid a cascade of spurious error messages resulting from a single error *Statistics101* will only display one error message per syntax check or per program run attempt. You should fix the reported error, then check the syntax or run the program again. If there are more errors, you will get another error message. If there are no more errors, the program will run to completion. For example, here is a short program with a spelling error.

```
COPY 1,10 name
STDEV namme sigma         'Misspelled "name"
PRINT sigma
```

Attempting to run this program produces this output:

```
ERROR: Input variable "namme" not previously defined in command STDEV,
Token[null], Edit line 2 (Debug line 2)
Program not run because of above error(s).
```

This error message is an indication that `namme` had not been given a value earlier in the program, therefore it can't provide a value to STDEV. *Statistics101* doesn't "know" that `namme` is misspelled, only that it hasn't been seen before and therefore can't serve as an input argument to a command. When you look at the line, you can see the spelling error.

When detecting spelling errors, *Statistics101* does not check your variable names against a dictionary as a word processor does. Instead, it checks your variable names against each other, so it doesn't matter how you spell them as long as when you're referring to the same variable you always spell it the same way. Therefore, in the above example, if “name” in the first line had been spelled “namme” it would have agreed with the “namme” in the second line and would not have caused an error message.

You probably noticed that the error message said that the error was on “Edit line 2 (Debug line 2)” and wondered about the apparent repetition. Since there is an edit window and a debug window in *Statistics101*, error messages specify which line number in each window is in error. If you don’t use INCLUDE files, the two line numbers will always match. If you use INCLUDE files, then the text displayed in the debug window will show all the included files (the edit window does not show the expanded INCLUDE files) and so a line at one position in the Edit window will be at a different position in the debug window. Thus the error message gives both positions.

Run-time Errors

Error messages during the execution, or “run-time,” phase are usually the result of logic errors in your simulation program. They arise from attempts to perform an arithmetic operation on an enum (a named constant that was not assigned a numerical value), or trying some operation on an unintentionally null vector. For these errors you’ll have to carefully review your *Resampling Stats* program’s logic. For example,

```
COPY 1,10 myList
'. . . other program commands left out for brevity . . .
WEED myList between 1 10 result
MULTIPLY result 5 answer
PRINT answer
```

is syntactically correct, but semantically incorrect because the variable `myList`, has become an empty vector. (WEED removes any members of its first argument that pass the subsequent test. In this case, any numbers between 1 and 10 are removed or “weeded” out.) Thus the program produces this error message at run time:

```
ERROR during execution of command MULTIPLY at Edit line 4 (Debug line 4)
java.lang.ArrayIndexOutOfBoundsException: -1
Program aborted.
```

The “-1” isn’t of much help; it’s the result of an array-bounds error (programmer lingo) resulting from trying to multiply by the empty vector, `result`. That’s the best *Statistics101* can do—it “knows” you told it to multiply by an empty vector, but must leave it to you to figure out why you did that. At least the main part of the error message tells you where to look for the cause of the error.

Tip: If you double-click on the ERROR line in the output window, the cursor in the Editor window will jump to and highlight the line containing the error.

If you have trouble finding a logic error in your simulation, *Statistics101*’s built-in debugger can help you find it. With the debugger, you can step through your program one command at a time looking at the contents of every constant and variable and even changing their values. You can set “breakpoints” to tell the program where to pause and switch into debug mode. You initiate the debug mode by clicking on the toolbar icon that looks like an insect. Read about how to use it in the *Statistics101* Help documentation. The help documentation is accessible via *Statistics101*’s Help>Help menu or by clicking on the “?” button in the lower right corner of the Debug window.

Tip: For a quick introduction to using the Statistics101 debugger you can watch the short video tutorial "Using the Debugger" at http://www.statistics101.net/FlashTutorials/FlashTutorialsList.html.

Part 3: Special Techniques

When programming a simulation, you might become aware of the need to perform some special calculation for which there is no *Resampling Stats* command. This section identifies several such special cases and shows how to accomplish them using the *Resampling Stats* language. It also presents subroutines that encapsulate the techniques so that they can be more easily used. Even if you don’t need to use these subroutines, reading this section will give you many programming concepts that you will find useful as you encounter other programming problems.

The subroutines introduced in this section (and many others) are stored in files in the *Statistics101 lib* folder so you can use them in your programs without having to copy and paste from this document. Each file contains one or more subroutines. To use a subroutine, you need to include its file at the top of your program using an INCLUDE command, like this:

```
INCLUDE "lib/nameOfFileContainingSubroutine.txt"
```

Make sure the "Enable auto-INCLUDE" checkbox is checked in the Preferences Editor tab (Edit>Preferences...>Editor). If it is, then after you type the name of a subroutine into your program, the next time you do a syntax check (Ctrl-K) or run the program (Ctrl-R), *Statistics101* will automatically add the appropriate INCLUDE commands to the top of your program.

If auto-INCLUDE is not enabled, then you have to enter any INCLUDE commands yourself. There are two ways that Statistics101 can make it easier for you to include a subroutine in your program:

If you know the name of the file containing the subroutine, right-click in the editor window and select the item “Include Lib file...” (or type Ctrl-U). This will bring up an open-file dialog. Select the desired file or files and click the Open button. This will add an INCLUDE command for each selected file to the top of your program.

If you don’t know the name of the file containing the desired subroutine, left-click in the editor at the start of a line to put the cursor where you want to invoke the subroutine. Then, hold down the Shift key while right-clicking anywhere in the edit window and select a subroutine name from the “Insert Subroutines From Lib By Name” submenu or from the “Insert Commands and Subroutines By Category” submenu. This will insert the name of the subroutine at the cursor and will also insert an INCLUDE command for the file containing that subroutine at the top of your program.

How to compare two vectors for equality

In some simulations, you might have to decide if two vectors are the same. Take a simple case: suppose you want to know the probability that if you flipped a coin five times, you would get exactly three heads followed by two tails. Here’s the beginning of the program:

```
CONST (1 2) heads tails 'Assign values so we can use arithmetic commands
COPY (heads tails) coin
COPY 1000 repeatCount
REPEAT repeatCount
   SAMPLE 5 coin outcome
   . . .
```

Now you've got your five coin flips in the `outcome` vector. How do you decide if it equals the desired result of three heads followed by two tails?

You might think that if you have two vectors with the same elements, and you subtract one from the other, and then sum all the differences, you would get a zero. That is true, but it is possible to get zero even if the vectors are different, as this example shows.

```
COPY (2 1 2 2 2) vec1
COPY (1 2 1 3 2) vec2
SUBTRACT vec1 vec2 difference
SUM difference diffSum
PRINT diffSum
```

The sum of the differences here is zero, yet the two vectors are different. That happened because some of the differences were negative and some were positive and they canceled out when you added them. You can still use subtraction if instead of using the simple differences you use the absolute value of the differences as in this example:

```
COPY (2 1 2 2 2) vec1
COPY (1 2 1 3 2) vec2
SUBTRACT vec1 vec2 difference
ABS difference absDiff
SUM absDiff absDiffSum
PRINT absDiffSum
```

Now the example prints that the absDiffSum is 4.0. With this revision, you are guaranteed that if the two vectors are equal then the absDiffSum will be zero and if they are not equal, the absDiffSum will not be zero.

This seems like a lot of work to go through just to compare two vectors. There is a simpler way. The *Resampling Stats* command SUMABSDEV performs the entire calculation for you. Here's the last example revised to use that command:

```
COPY (2 1 2 2 2) vec1
COPY (1 2 1 3 2) vec2
SUMABSDEV vec1 vec2 absDiffSum
PRINT absDiffSum
```

Now you can use this technique in your five coin flip simulation as follows.

```
CONST (1 2) heads tails
COPY (heads tails) coin
COPY 0 successCount
COPY 10000 repeatCount
REPEAT repeatCount
   SAMPLE 5 coin trial
   SUMABSDEV (heads heads heads tails tails) trial result
   IF result = 0
      ADD 1 successCount successCount
   END
END
DIVIDE successCount repeatCount probability
PRINT probability
```

If your vectors might have enums in them, then the above technique will not work. That's because SUMABSDEV only accepts numbers and Named Values. For example, if I had written

```
ENUM heads tails
```

```
COPY (heads tails) coin
...
```

then there would have been an error message when SUMABSDEV tried to do its work. Instead, you can compare any two vectors using the following subroutine. This subroutine works for vectors that have any combination of enums, Named Values, and/or floating point numbers. You will find this subroutine in *Statistics101*’s *lib* directory in the file *lib/vectorCommands.txt*.

```
NAME (0 1) equal notEqual 'Define constants equal and notEqual.
GLOBAL equal notEqual

'Compares two vectors element by element for equality.
'Returns equal (0) if they are equal.
'Returns notEqual (1) if they are not equal.
'If their sizes differ they are not equal.

NEWCMD EQUAL_VECTORS vec1 vec2 result ?"Returns 0 if vec1 = vec2."
   SIZE vec1 vec1Size
   SIZE vec2 vec2Size
   COPY equal result
   IF vec1Size <> vec2Size
      COPY notEqual result
   ELSE
      FOREACH pos 1,vec1Size
         TAKE vec1 pos v1element
         TAKE vec2 pos v2element
         IF v1element <> v2element
            COPY notEqual result
            BREAK
         END
      END
   END
END
```

How to determine if all the elements of a vector are equal

In other situations, you might need to know if all the elements of a vector are the same. For example, what is the probability that, if you flip a coin five times, all five flips will have the same result, i.e., all heads or all tails?

Using the technique just discussed, you can enlist SUMABSDEV for this purpose as follows:

```
CONST (1 2) heads tails
COPY (heads tails) coin
COPY 0 successCount
COPY 10000 repeatCount
REPEAT repeatCount
   SAMPLE 5 coin trial
   SUMABSDEV (heads heads heads heads heads) trial result1
   SUMABSDEV (tails tails tails tails tails) trial result2
   IF result1 = 0 OR result2 = 0
      ADD 1 successCount successCount
   END
END
DIVIDE successCount repeatCount probability
PRINT probability
```

Here you are doing two comparisons, one for each desired outcome. But what if the problem were more complicated? Say that you were rolling five dice and wanted to know the probability that all five rolls came up the same. Using the above technique you would have six OR terms in the IF command's logical expression—one for each die face. Instead, you can take advantage of the fact that *Resampling Stats*/*Statistics101* will automatically extend a vector by repeating the last element. So if you take the first element of a vector and use it to compare with the entire vector, you have the test you are looking for. Here's the program.

```
COPY 1,6 die
COPY 0 successCount
COPY 1000000 repeatCount
REPEAT repeatCount
   SAMPLE 5 die trial
   TAKE trial 1 firstElement
   SUMABSDEV firstElement trial result
   IF result = 0
      ADD 1 successCount successCount
   END
END
DIVIDE successCount repeatCount probability
PRINT probability
```

The trick here is: if all the elements of a vector are the same and you choose the first (or any other) element, and extend it to have the same number of elements as the original vector, then both vectors will be identical.

If your vector might have enums in it, then the above technique will not work. That's because SUMABSDEV only accepts numbers and Named Values. Instead, you can test it using the following subroutine. This subroutine works for vectors that have any combination of enums, Named Values, and/or floating point numbers. You will find this subroutine in *Statistics101*'s *lib* directory in the file *lib/vectorCommands.txt*.

```
NEWCMD EQUAL_ELEMENTS vec result ?"Returns 0 if all vec's elements are equal"
   result = equal
   TAKE vec 1 firstElement
   FOREACH element vec
      IF element <> firstElement
         result = notEqual
         BREAK
      END
   END
END
```

How to detect a sequence

Sometimes you might have to determine whether a vector contains a sequence of numbers, such as (3 4 5 6 7). To do that in *Resampling Stats* you can make use of the fact that if you copy a vector that has sequential elements and shift its contents by one, take the difference, then remove the first and last elements of the result, the elements of the resulting vector will all be equal. For example,

```
(3  4  5  6  7)      ' original sequence
(0  3  4  5  6  7)   ' shifted sequence
(3  1  1  1  1  0)   ' difference
(1  1  1  1)         ' after removing first and last elements
```

Once you've done the shift, subtract, and remove, you can test as shown in the previous section to see if all the elements of the result are equal.

There are two ways to shift the elements of a vector. One way uses the COPY command's concatenation capability. Here I use a COPY command to shift a zero onto the left of a vector:

```
COPY 0 vec shiftedVec  'shift vec one place to the right; fill with zero
```

The second way is to use the SHIFT command, like this:

```
SHIFT 1 vec shiftedVec  'shift vec one place to the right; fill with zero
```

There is one difference between the results of these two commands. COPY adds a zero to the front of the vector, therefore `shiftedVec` is larger by one than `vec`. In contrast, while SHIFT also adds a zero to the front of the vector, it makes room for it by dropping the rightmost element. Therefore with the SHIFT command `shiftedVec` is the same size as `vec`.

Here's a program that uses all the special techniques discussed so far. It computes the probability of a poker straight including the probability of a straight flush. The bold lines are the specific commands that decide if a sequence has occurred.

```
'Compute probability of a poker straight:
'A Straight consists of 5 cards with consecutive values with the
'addition that Ace can be high (13) or low (1) as needed
'to complete the sequence. (This includes straight flushes.)
COPY 100000 repeatCount
COPY 0 straightCount
COPY 1,13 1,13 1,13 1,13 deck
REPEAT repeatCount
   SHUFFLE deck shuffledDeck
   TAKE shuffledDeck 1,5 hand
   SORT hand sortedHand
   COPY 0 sortedHand shiftedSortedHand
   SUBTRACT sortedHand shiftedSortedHand diffVector
   TAKE diffVector 2,5 diffVectorCenter
   COUNT diffVectorCenter = 1 diffVectorSum
   IF diffVectorSum = 4
      ADD 1 straightCount straightCount
   END
   '  Handle special case where ace = 13:
   SUBTRACT sortedHand (1 9 10 11 12) aceWild
   COUNT aceWild  =0 aceWildSum
   IF aceWildSum = 5 'we have a match
      ADD 1 straightCount straightCount
   END
END
DIVIDE straightCount repeatCount probability
PRINT probability
```

Since there are a number of commands involved, you can encapsulate the idea into a subroutine that will test whether a vector's elements constitute a sequence. Here's one way to do it using the SUMABSDEV technique described in the last section to test whether all of diffVectorCenter's elements are equal to one (If you want to test a vector as-is, without sorting, remove the SORT command from the subroutine.):

```
'Subroutine tests vec to determine if it is an arithmetic
'sequence all of whose elements differ by one when sorted.
```

```
'If vec is such a sequence, result will be true (1).
'Otherwise, result will be false (0).
'
NEWCMD IS_SEQUENCE vec result @series @"vector operations" \
?"Returns true if vec is an arithmetic sequence with common difference
between elements of one, else false."
   SORT vec sortedVec
   COPY 0 sortedVec shiftedSortedVec
   SUBTRACT sortedVec shiftedSortedVec diffVector
   SIZE vec vecSize
   TAKE diffVector 2,vecSize diffVectorCenter
   SUMABSDEV 1 diffVectorCenter dev
   IF dev = 0
      result = true
   ELSE
      result = false
   END
END
```

Note the use of the predefined named constants, true and false. These were discussed earlier. Now, all you have to do to see if a vector contains a sequence is use the new one-line command, IS_SEQUENCE.

```
' Assume myVec is defined earlier in the program.
IS_SEQUENCE myVec result
IF result = true
' do whatever needs to be done for a sequence
ELSE
' do whatever needs to be done for a non-sequence
END
```

How to sort related vectors

If you have data for different properties of items in the population and you want to analyze it or resample from it, you might have to keep the properties for each item aligned when you sort one of them. For example assume you have the following data representing five different individuals:

```
ENUM male female
DATA (62 68 73 58 66) height
DATA (120 165 198 99 115) weight
DATA (female male male female female) sex
```

Say you wanted to sort the data by height. If you just apply the SORT command to the height vector, the data in the other vectors will no longer correspond to the height numbers. If you apply SORT to the others, they will become even more uncoordinated. What you want is to sort the height vector, then reposition the elements of the other vectors to the positions that their corresponding height took on after the sort. You do this using the TAGSORT command to obtain the position numbers for each element of the height vector as if it were sorted. Then you use the TAKE command to reposition the elements of all the vectors to that order. Here's how:

```
TAGSORT height heightTags
TAKE height heightTags sortedHeight
TAKE weight heightTags sortedWeight
TAKE sex heightTags sortedSex
PRINT sortedHeight sortedWeight sortedSex
```

This produces the following output:

```
sortedHeight: (58.0 62.0 66.0 68.0 73.0)
sortedWeight: (99.0 120.0 115.0 165.0 198.0)
sortedSex: (female female female male male)
```

You could generalize this into a subroutine that would do the sort as follows.

```
'Subroutine to do a coordinated sort, in place, of three vectors
'based on the first vector (keyVec).
NEWCMD SORTCOORD keyVec vec2 vec3 ?"Coordinated sort of three vectors"
   TAGSORT keyVec tags
   TAKE keyVec tags keyVec
   TAKE vec2 tags vec2
   TAKE vec3 tags vec3
END
```

Here is a program that uses this subroutine to perform three coordinated sorts, one on each vector:

```
SORTCOORD height weight sex
PRINT height weight sex
PRINT                         'print a blank line
SORTCOORD weight height sex
PRINT height weight sex
PRINT
SORTCOORD sex weight height
PRINT height weight sex
```

And here is its output:

```
height: (58.0 62.0 66.0 68.0 73.0)
weight: (99.0 120.0 115.0 165.0 198.0)
sex: (female female female male male)

height: (58.0 66.0 62.0 68.0 73.0)
weight: (99.0 115.0 120.0 165.0 198.0)
sex: (female female female male male)

height: (68.0 73.0 58.0 66.0 62.0)
weight: (165.0 198.0 99.0 115.0 120.0)
sex: (male male female female female)
```

Tip: If you frequently need to do coordinated sorting with different numbers of arguments, you can use the subroutines defined in the file lib/coordinatedVectorCommands.txt. That file includes two commands that do coordinated sorts for any number of vectors. One (SORTCOORD) does an ascending sort; the other (SORTCOORD_DESC) does a descending sort. Just add the command

```
INCLUDE "lib/coordinatedVectorCommands.txt"
```

at the top of your program and then you can invoke whichever of those subroutines you need.

How to shuffle related vectors

Instead of sorting related data, you might want to randomize them for when you are simulating sampling without replacement. You can apply some of the ideas from the previous sorting technique. Assuming the same dataset we used above, you first create a vector that plays the role of the tag vector in the previous technique. It should be the same size as the other vectors, because it has one number for each position. Since the example vectors have five elements, this position vector will have five elements.

```
COPY 1,5 positions
SHUFFLE positions shuffledPositions
TAKE height shuffledPositions shuffledHeight
TAKE weight shuffledPositions shuffledWeight
TAKE sex shuffledPositions shuffledSex
PRINT shuffledHeight shuffledWeight shuffledSex
```

One run produced the following output:

```
shuffledHeight: (66.0 73.0 68.0 62.0 58.0)
shuffledWeight: (115.0 198.0 165.0 120.0 99.0)
shuffledSex: (female male male female female)
```

Multiple runs will produce different orders, but the related elements will always be moved together to their shuffled positions. This can be generalized into a subroutine as follows:

```
'Subroutine to do a coordinated shuffle, in place, of three vectors.
NEWCMD SHUFFLECOORD vec1 vec2 vec3 ?"Coordinated shuffle of three vectors"
   SIZE vec1 vecSize
   COPY 1,vecSize positions
   SHUFFLE positions positions
   TAKE vec1 positions vec1
   TAKE vec2 positions vec2
   TAKE vec3 positions vec3
END
```

Note the use of the SIZE command to determine the number of elements in the vectors. This then allows the creation of the position vector in the next command. The next program segment shows an invocation of the subroutine:

```
SHUFFLECOORD height weight sex
PRINT height weight sex
```

with the following output:

```
height: (73.0 62.0 68.0 66.0 58.0)
weight: (198.0 120.0 165.0 115.0 99.0)
sex: (male female male female female)
```

Each invocation of SHUFFLECOORD will produce different orders, but all the related elements will be coordinated.

Tip: If you frequently need to do coordinated shuffling with different numbers of arguments, you can use a subroutine defined in the file lib/coordinatedVectorCommands.txt. That file includes the command SHUFFLECOORD, which does coordinated shuffles for any number of vectors. Just add the command

```
INCLUDE "lib/coordinatedVectorCommands.txt"
```

at the top of your program and then you can invoke the SHUFFLECOORD command where you need to in your program.

This coordinated shuffling technique (without using a subroutine) is used in the next program to compute the probability of a poker straight excluding straight flushes. The boldfaced lines are the ones using this coordinated shuffle technique.

```
'Compute probability of a poker straight excluding straight flushes:
'A Straight consists of 5 cards with consecutive values with the
'addition that Ace can be high (13) or low (1) as needed
'to complete the sequence. A flush is 5 cards of the same suit.
'
'Since the card values and suits have to be considered separately
'and yet the correlation between suits and values is important for
'this problem, a third vector, "deckPositions", is used to
'coordinate the related suits and values.

COPY 1000000 repeatCount
ENUM heart club diamond spade
COPY 1,13 1,13 1,13 1,13 deckValues
COPY 13#heart 13#club 13#diamond 13#spade deckSuits
COPY 1,52 deckPositions
COPY 0 straightCount
REPEAT repeatCount
   SHUFFLE deckPositions shuffledPositions
   TAKE shuffledPositions 1,5 handPositions
   TAKE deckValues handPositions handValues
   SORT handValues sortedHandValues
   'Shift vector right by one position:
   COPY 0 sortedHandValues shiftedSortedHandValues
   SUBTRACT sortedHandValues shiftedSortedHandValues diffVector
   'Remove the 1st and last elements:
   TAKE diffVector 2,5 diffVectorCenter
   COUNT diffVectorCenter = 1 diffVectorSum
   IF diffVectorSum = 4
      'We have a straight. Check if suits match.
      TAKE deckSuits handPositions handSuits
      TAKE handSuits 1 firstSuit
      SUMABSDEV handSuits firstSuit result
      IF result <> 0
         ADD 1 straightCount straightCount
      END
   END
   'Handle special case where ace = 13:
   SUMABSDEV sortedHandValues (1 9 10 11 12) aceWild
   IF aceWild = 0
      'We have an Ace high straight. Check if suits match.
      TAKE deckSuits handPositions handSuits
      TAKE handSuits 1 firstSuit
      SUMABSDEV handSuits firstSuit result
      IF result <> 0
         ADD 1 straightCount straightCount
      END
   END
END
DIVIDE straightCount repeatCount probability
PRINT probability
```

How to filter related vectors

If you have a set of related data like the height, weight, and sex data seen earlier, you might want to select a subset that meets certain criteria. For example, say you wanted the subset consisting of all individuals whose height is less than or equal to 65 inches. For this, you can use the TAGS command as follows. The data declarations are repeated here for convenience:

```
ENUM male female
DATA (62 68 73 58 66) height
DATA (120 165 198 99 115) weight
DATA (female male male female female) sex

TAGS height <=65 heightTags
TAKE height heightTags filteredHeight
TAKE weight heightTags filteredWeight
TAKE sex heightTags filteredSex
PRINT table filteredHeight filteredWeight filteredSex
```

The TAGS command puts the position numbers of all the elements of `height` that are less than or equal to 65 into the vector `heightTags`. Then those tags are used to copy the elements from those same positions in each related input vector into a new output vector. The job of the TAKE commands can be generalized into a subroutine that will allow you to filter any number of input vectors. Here is how that can be done.

```
NEWCMD TAKECOORD tags #"{inVector outVariable}"
   ARGCOUNT numberOfArgs
   REMAINDER numberOfArgs 2 remainder ' must be an even number of arguments.
   IF numberOfArgs >= 2 AND remainder = 0
      COPY 1 argNum
      WHILE argNum < numberOfArgs
         GETARG argNum inArg
         ADD 1 argNum argNum
         GETARG argNum outArg
         TAKE inArg tags outArg
         ADD 1 argNum argNum
      END
   ELSE
      DEBUG "ERROR: Incorrect number of arguments in TAKECOORD."
   END
END

TAGS height <=65 heightTags  'Perform test prior to invoking subroutine
TAKECOORD heightTags height filteredHeight weight filteredWeight \
sex filteredSex
PRINT table filteredHeight filteredWeight filteredSex
```

Above, I first defined the subroutine, TAKECOORD. Then I used the TAGS command to generate a list of the positions of the elements that are less than or equal to 65. Finally, I used those tags as input to the TAKECOORD subroutine to obtain the answer. The TAKECOORD subroutine is in the subroutine library in the file coordinatedVectorCommands.txt so you can use it by merely invoking it.

In both cases, the result is as follows:

```
filteredHeight    filteredWeight    filteredSex
62                120               female
58                99                female
```

Tip: If you frequently need to do coordinated filtering with different numbers of arguments, you can use another subroutine defined in the file lib/coordinatedVectorCommands.txt. That subroutine is called SELECTCOORD. It combines the TAGS command and the TAKECOORD command into one subroutine to do coordinated filtering for any number of vectors based on a test you supply. You can use the Subroutine Browser to find and read its description and definition.

How to iterate over related vectors

The FOREACH command provides an easy way to iterate over a single vector as you saw earlier. But suppose you had several related vectors such as the height, weight, and sex data that we've been discussing, and that you wanted to know how many men under 65 inches weighed more than 200 pounds. In this case, you have to iterate over the vectors in a coordinated way testing the elements of all the vectors at each position. Here's a program that would do the job:

```
ENUM male female
DATA (62 68 73 58 66 60 58) height
DATA (120 165 198 99 115 210 340) weight
DATA (female male male female female male male) sex

'Count the number of men under 65 inches who
'weigh more than 200 pounds.
COPY 0 overweightMen
SIZE height vecSize
FOREACH position 1,vecSize
   TAKE height position heightElement
   TAKE weight position weightElement
   TAKE sex position sexElement
   IF sexElement = male AND heightElement < 65 AND weightElement >200
      ADD 1 overweightMen overweightMen
   END
END
PRINT overweightMen
```

Here a sequence is created containing the numbers from 1 to the size of one of the data vectors. (All the data vectors must be the same size.) The FOREACH command iterates through that sequence, assigning each number in turn to the variable named "position." That variable is used to TAKE the elements at the same position from all the data vectors. Then the elements are tested with the IF command and if the elements all satisfy the test criteria, the count of overweight men is incremented.

How to avoid the "Out of Memory" error

If your *Resampling Stats* program has very large vectors, it may run out of memory space. If that happens, you will get an error dialog notifying you of the condition, and the program will abort. Most of the time, you can revise your program so that it does not require so much memory. For example, here is our familiar "two heads out of three tosses" coin problem, with an unnecessarily large number of repetitions as a demonstration:

```
ENUM heads tails
COPY heads tails coin
COPY 10000000 numberOfTrials
REPEAT numberOfTrials
   SAMPLE 3 coin trial      ' flip coin 3 times
   COUNT trial = heads headcount
   SCORE headcount results
```

```
END
COUNT results = 2 successes
DIVIDE successes numberOfTrials probability
PRINT probability
```

When you run this program, you will get the "Out of memory" message and the program will abort (unless you have a huge amount of RAM allocated to your Java virtual machine). The reason the program consumes so much memory is that it uses the SCORE command to save the results of each trial. Therefore, the results vector grows to contain `numberOfTrials` elements. In simulating this problem, there is no reason to save all those results since all we want is the count of successes. This program can be revised so that it will never overflow memory, like this:

```
ENUM heads tails
COPY heads tails coin
COPY 10000000 numberOfTrials
COPY 0 successes
REPEAT numberOfTrials
   SAMPLE 3 coin trial     ' flip coin 3 times
   COUNT trial = heads headcount
   IF headcount = 2
      ADD 1 successes successes
   END
END
DIVIDE successes numberOfTrials probability
PRINT probability
```

The revised program simply counts the number of successes directly, without accumulating them in a vector. It also probably runs slightly faster than the earlier one.

The general approach to avoiding "out of memory" errors is to look for arrays that grow during the simulation and try to revise the simulation to minimize or eliminate those arrays or their growth.

Bibliography

Downing, Douglas, Ph.D., and Jeffrey Clark, Ph.D. *Statistics The Easy Way*, Barron's Educational Series, Inc., 1997.

Freedman, David H., Wrong: Why experts keep failing us--and how to know when not to trust them, Little, Brown and Company Hachette Book Group, 2010.

Mooney, Christopher Z., and Robert D. Duval, *Bootstrapping: A Nonparametric Approach to Statistical Inference* , Sage Publications, Newbury Park, California, 1993.

Mooney, Christopher Z., *Monte Carlo Simulation* , Sage Publications, Thousand Oaks, California, 1997.

Simon, Julian, *Resampling: The New Statistics*, Resampling Stats, Inc., Arlington, Virginia, http://www.resample.com/intro-text-online/.

Voelker, David H., MA et al., *CliffsNotes Statistics Quick Review* , Hungry Minds, Inc. New York, NY, 2001.

Appendix 1: Glossary

Actual Argument Any argument listed on a subroutine invocation. Contrast with "Dummy Argument."

Annotation Annotations are optional markups on a subroutine that provide information about the subroutine that becomes integrated into Statistics101 help system. There are two types of annotations: 1. A description annotation, which is a one-line description of the purpose of the subroutine. 2. A category annotation, which specifies the category or categories that the subroutine's name should be listed with in the Subroutine Browser and the Edit Window popup menu.

Argument An input or an output of a command or a subroutine. An argument may be a constant or a variable.

Bootstrap A resampling method that uses a set of sample data as if it were the population under study by taking random samples with replacement from the original sample and using these derived samples to compute estimates of the distributions of desired test statistics.

Command The unit of computation of Resampling Stats and Statistics101. Each command operates on data and produces a result. Each command is normally confined to one text line with the command's name being first on the line followed by zero or more keywords, followed by zero or more arguments. A command may be extended over multiple lines if the backslash continuation character ("\") is the last character on each line but the last.

Comment An explanatory note in a Statistics101 program that is intended for human readers, not the computer. There are two kinds of comments. The first is a one-line comment that is introduced by a single quote character. The second type is the block comment, which can span multiple lines and is enclosed between two markers, like this: /* block comment */.

Compound Command A command that includes several commands in its scope. Examples are IF, NEWCMD, REPEAT, FOREACH, UNTIL, and WHILE. The scope of a compound command is terminated by an END command.

Compound Test One or more simple tests combined with logical operators. For example:

```
number1 <= number2 AND vecA between 1 10.
```

Compound tests are allowed only in the IF, ELSEIF, UNTIL, and WHILE commands.

Conditional Loop A loop whose number of repetitions is not fixed but is determined by a logical expression. WHILE and UNTIL loops are conditional loops. REPEAT and FOREACH are "unconditional" loops.

Constant 1. A vector whose content does not change, usually a literal vector, such as (1 4.5 2 6.7 9). 2. A number that is given a name by the CONST or NAME command. 3. A string literal. Contrast with “Variable.”

Declaration 1. The introduction of a subroutine’s name and argument list into a program. The NEWCMD command declares a subroutine as a byproduct of defining the subroutine. The DECLARE command declares but does not define the subroutine. (See also “Definition” and “Invocation.”) 2. The introduction of a vector name into a program as the output argument of some command. For example,

```
COPY vec1 vec2
```

introduces vec2 into the program, assuming that this is its first appearance. vec1 is an input and therefore must have been declared previously.

Definition The complete description of a subroutine including all the commands between its NEWCMD command and its END command. See also “Declaration” and “Invocation.”

Dummy Argument Any name listed as a subroutine argument in a NEWCMD command. A dummy argument is just a placeholder. When a subroutine is invoked, it is invoked with actual arguments that will be substituted for the dummy arguments based on their positions in the argument list. The actual arguments need not have the same names as the dummy arguments they are replacing.

Element Any individual number or named constant in a vector.

Enum Short for “enumerator.” A named constant that was not given a value by the user. It has a hidden arbitrary value assigned by the ENUM or NAME command. Since it does not have a meaningful numerical value, it cannot be used with arithmetic commands such as ADD, SUBTRACT, SUMABSDEV, SQUARE, etc. Contrast with “Named Value.” Example:

```
ENUM mon tue wed thu fri sat sun
```

Global Name A name of a variable or constant vector or Named Constant or String that is visible (see Visibility) throughout a program, including within subroutines. Names are made global being listed in a GLOBAL command. The GLOBAL command cannot be used within a subroutine. Compare with “Local Name” and “Non-local Name.”

Invocation The use of a subroutine in a program. (Contrast with “Declaration” and “Definition.”)

Keyword A predefined word added to a command’s argument list that modifies the behavior of the command. For example, the keyword table directs the PRINT command to print its arguments in a table format instead of the usual horizontal format. Some keywords may take an argument themselves. For example the binsize keyword of the HISTOGRAM command takes a number as an argument.

List A one-dimensional ordered collection of numbers considered as a unit. Usually refers to a literal vector. See the entry for Vector in this glossary.

Literal 1. A number without a name, e.g., 5.789. 2. A vector without a name, e.g., (1 5 3 4 ace 2 7). 3. A text string without a name. Keywords, named constants, and the missing numbers codes (period and NaN) are also considered to be literals. (See also Missing Number)

Local Name A name of a variable or constant vector that is visible (see Visibility) only within a subroutine. The name of any vector or string declared within a subroutine is a local name. Compare with "Global Name" and "Non-local Name."

Logical Expression A logical expression is a comparison between values that produces a true or false result. There are two types of logical expressions, simple tests or just tests (e.g., number1 < number2) and compound tests. See "compound test."

Missing Number A special literal that represents the fact that for that position in a vector, the data is missing. A missing number is represented in Statistics101 by either a period (".") or the characters "NaN", which is an acronym for "Not a Number." For example, if you had a set of data of students' height versus weight, but for some students all you had was their height or their weight, you would mark the missing data with NaN or period like this:

```
DATA (62 71 60 68 NaN 72) height
DATA (102 115 . 170 145 194) weight
```

With regard to tests on missing data (NaN or "."), NaN is equal only to NaN, not to any other number. Furthermore, NaN is not greater than or less than any number, including NaN.

Mixed Vector A vector that contains both numerical values and Named Constants. Example: (ace deuce trey 4 5 6 7)

Model The detailed representation of a probability or statistical simulation in the form of a Resampling Stats program.

Monte Carlo Simulation The use of random or pseudo-random data as input to a computer model of a process to obtain results that are difficult or impossible to characterize with a mathematical formula. The name comes from the capital of Monaco, known for its casinos. Resampling methods are Monte Carlo simulations.

Multiple Specification A combination of an operator, consisting of the pound-sign (#) and two arguments, one each on its left and right sides that specifies a vector. The general form of the multiple specification is n1#vec, where n1 is a number (see the definition of number) that determines how many times all the elements of the vector vec will be reproduced in the resulting vector. vec may be any kind of vector specification, including a Sequence specification, except that it may not be another Multiple specification. For example, 3#(1 2 3) specifies the vector (1 2 3 1 2 3 1 2 3). The specification 3#1,3 specifies the same vector. Both arguments of the Multiple specification may be variables. The Multiple specification may be used anywhere a literal vector is allowed in the language except, as just noted, in another Multiple specification.

NaN A literal representing a missing number. NaN is an acronym for “Not a Number.” Typographical case doesn’t matter, so “NAN” or any other combination of upper and lower case is equivalent to NaN. A missing number can also be represented by a period (“.”). See Missing Number.

Named Constant Any name defined by the ENUM, CONST or NAME commands. There are two kinds of Named Constants: enums and Named Values.

Named Value A Named Constant that was given a value by the user in a CONST or NAME command. Since these have meaningful numerical values, they can be used with arithmetic commands such as ADD, SUBTRACT, SUMABSDEV, SQUARE, etc. Contrast with “Enum.” Example:

```
CONST (2.71828 3.1459) e pi
```

Nesting The inclusion of a compound command within another compound command. See “Compound Command.”

Non-Local Name A name of a variable or constant that is visible (see Visibility) only outside of subroutines, not within subroutines. Non-local names are names declared outside of subroutines, that are not declared to be GLOBAL. Compare with “Local Name” and “Global Name.”

Number 1. A literal value such as 1.23 2. A vector with only one element. 3. A vector with more than one element but of which only the first will be used (as in a test such as vecA > 5).

Operator A keyword or symbol that performs or describes a computation. Comparison operators, such as =, <, and “between” are used in simple tests. Logical operators, such as AND, OR, and NOT are used in compound tests. Arithmetic and concatenation operators, such as +, -, and & are used in the LET command. The generative operators, # and comma (for example, as in 3#5 and 1,10,2), are used to generate vectors with repetitive or sequential elements.

Optional Argument An optional dummy argument of a subroutine. Optional arguments are introduced by the “#” keyword. As an example, here is a subroutine declaration that allows dummy arguments:

```
NEWCMD MYSUB inputArg #"{optionalArgs}"
...
END
```

This declares that MYSUB takes one input argument and zero or more optional arguments.

Precedence The order in which operators will be evaluated in cases where a decision must be made between two or more operators.

Probability For the purposes of deriving probabilities by simulation, probability is defined as the relative frequency of a random experiment's outcomes. In other words, for a particular result of a simulation experiment, the probability is the ratio of the number of times that that result occurred to the number of times the experiment was run. As a formula,

Probability = number of "successes" / number of trials.

Process The origin of the randomness implicit in a probability problem. The "process" is the method you use to select your sample from the population and to separate the desired outcomes from the undesired outcomes.

Program An ordered sequence of commands designed to achieve some computational result.

Resampling The use of the observed data or of a data generating mechanism (such as a die or a computerized random number generator) to produce new hypothetical samples, the results of which can then be analyzed. (after Simon's Resampling: The New Statistics, p. 2)

Resampling Stats The name of the statistical simulation language originally developed by Julian Simon, Peter Bruce, Dan Weidenfeld and others. Statistics101 executes a greatly enhanced version of the language.

Runtime Error An error that is detected by Statistics101 during the execution of a Resampling Stats program. These are errors in the logic of your program. The Statistics101 debugger can help you understand and fix the cause. Contrast with "syntax error."

Scientific Notation A way of representing any number as a number between 1 and 10 (called the "mantissa" or "significand") multiplied by 10 to an exponent. For example, the number 12345.67 is written as 1.234567E4. The "E4" indicates that the 1.234567 must be multiplied by 10 to the fourth power, or 104. The mantissa may be positive or negative and the exponent may be positive or negative. In Resampling Stats, however, plus signs are not allowed. Numbers are assumed positive unless preceded by a minus sign.

An easy way to translate from scientific notation to a normal ("floating point") number is this: for positive exponents, move the decimal point of the mantissa to the right by the number of places specified by the exponent. For negative exponents, move the point to the left by the number of places specified by the exponent.

Sequence Specification A combination of an operator, consisting of one or two commas, and two or three numbers, that specifies an increasing or decreasing sequence of numbers. The general form of the sequence specification is n1,n2,n3 where n1 is the starting number (see the definition of number) for the sequence, n2 is the ending number of the sequence, and n3 is the size of the step between any two adjacent numbers. The ending number, n2, may be less than n1. In that case the sequence is in descending order. The step size, n3, is optional and if present must be greater than zero (may not be negative). If it is omitted (along with its preceding comma), the step size defaults to one. As an example, the sequence specification 1,100 specifies a sequence consisting of all the numbers from 1 to 100, inclusive. A second example, 1,100,2 specifies all the odd numbers between 1 and 100. In this case, the value 100 is not a member of the sequence. Any or all arguments of the Sequence specification may be variables. The sequence specification may be used anywhere a literal vector is allowed in the language.

Simulation Duplication of the important characteristics of a process or phenomenon, usually using a computer, to discover consequences of the process or phenomenon.

Statistics101 The name of the computer application that is the subject of this document. Statistics101 executes programs written in the Resampling Stats language. Together, the pair solve problems, including problems in probability and statistics by simulation, typically using pseudorandom numbers.

String A term referring to a string of text, i.e., a sequence of characters. There are two kinds of strings in Resampling Stats: 1. A “string literal” or “literal string” which is some text enclosed between double quotation marks. 2. A “string variable” or “variable string,” which is a named variable created with the STRING command. Either may be used wherever a command calls for a text string.

Subroutine A named set of commands that is treated as a unit. Subroutines are created using the NEWCMD command.

Syntax Error The result of an invalid construct in a Resampling Stats program. Typical syntax errors include a misspelled command, a command with an incorrect number of arguments, a misspelled variable name, etc. Contrast with “runtime error.”

Test A comparison among values yielding a result of true or false. For example, vec1 <= vec2 tests whether the first element of vec1 is less than or equal to the first element of vec2.If it is, the test evaluates to true, otherwise, to false. Also called a “simple test.” See also, “compound test.”

Type Named Constants created by the same ENUM, CONST, or NAME command are of the same “type.” Named constants created by separate ENUM, CONST, or NAME commands are of different “types” unless the TYPE keyword is used to specify type commonality. Named Constants of the same type sort together. Two Named Constants of different types are never equal to each other.

Variable A named vector or named string whose value is allowed to change during the execution of a program. Contrast with “Constant.”

Vector A one-dimensional ordered collection of numbers considered as a unit. The vector may be a variable or a constant and may or may not have a name. In Resampling Stats, a vector is represented as a sequence of numbers and/or Named Constants separated by spaces and enclosed in parentheses. For example: (2.3 4 6.7 -4 21). “List” and “vector” are used interchangeably in this document.

Visibility Visibility refers to the portions of a program wherein the name of a variable or constant may be used relative to where it is declared. In Statistics101 a name is visible only after the name is declared (see “declaration”). There are four categories of visibility:

Dummy: Any subroutine argument (“dummy variable”) name. The names of dummy variables are visible only within their subroutine.

Local: Any variable or constant name declared within a subroutine, that is not a dummy variable. A local name is visible only within that subroutine.

Non-local: Any variable or constant name declared outside a subroutine. Non-local names are visible at any point after their creation except within subroutines.

Global: Any variable or constant name declared to be global by a GLOBAL command. Global names are visible anywhere in the program following the GLOBAL command. They are visible both within and outside of subroutines. (The GLOBAL command itself cannot be used inside a subroutine.)

Appendix 2: Complete Command/Subroutine Descriptions

This table lists and describes the commands and subroutines supplied in the *Statistics101* lib directory. The command names are shown in normal ("roman") type; the subroutine names are shown in oblique ("italic") type. You can find complete information on each command in the "help" documentation which is available by selecting the Help>Help menu item. You can find full information on each subroutine, by using the subroutine browser, which is available by selecting the Window>Show Subroutine Browser menu item.

The easiest way to find a command or subroutine when you don't know its exact name or if it even exists is to do a keyword search using the Command/Subroutine Index, which is available from the *Statistics101* program's Help menu, Help>Command/Subroutine Index or by pressing the F3 function key.

To use any of the subroutines you must add an INCLUDE command to your program, that identifies the file containing the subroutine. For example, for the BETA subroutine, you would use the command:

```
INCLUDE "lib/BetaDistribution.txt"
```

If you have enabled the "Auto-INCLUDE" feature in the "Edit>Preferences...>Editor" dialog, then the necessary INCLUDE commands will be added automatically when your program's syntax is checked or the program is run.

Also in the lib directory are some files that define some commonly used math and logical constants. The file *lib/mathConstants.txt* contains definitions for constants such as *pi*, *e*, and conversions between radians and degrees. You can INCLUDE this file in your program to make the constants available. This file is not "auto-Included," so you'll have to put the INCLUDE command in manually. The easiest way to include it is to select the Edit>Include Lib File(s)... menu item (or press Ctrl-U) then choose the mathConstants.txt file from the Open File dialog that appears. When you click the Open button, an INCLUDE command for that file will be added to the top of your program.

ABS
Computes the absolute value of each element of the input vector.

ACOS
Computes the arc cosine of each element of its input vector.

ACOSDEG
Computes the arc cosine, in degrees, of each element of its input vector.

ADD
Arithmetically adds corresponding elements of its input vectors.

ARGCOUNT
Returns the number of arguments that its enclosing subroutine had when invoked.

ASIN
Computes the arc sine of each element of its input vector.

ASINDEG

Computes the arc sine, in degrees, of each element of its input vector.

ATAN
Computes the arc tangent of each element of its input vector.

ATANDEG
Computes the arc tangent, in degrees, of each element of its input vector.

AUTOCORR
Computes a vector of autocorrelations of the input vector for all time lags (position shifts).

AUTOCORRGRAPH
Displays a graph of the autocorrelation of the given vector at all lag intervals.

BETA
Generates a given number of random numbers from an approximation to the Beta Distribution, Beta(a,b) for integers a and b >= 1 with result being in the range from min to max.

BETA_01
Generates a given number of random numbers from an approximation to the Beta Distribution, Beta(a,b) for integers a and b >= 1 with result being in the range from 0 to 1.

BINOMIALPROB
Computes the probability of k successes in n trials given a probability of success.

BINOMIALSAMPLE
Selects a given number of samples from a binomial distribution.

BINOMIALTRIALS
Generates numberOfTrials samples of size sampleSize and records the number of successes from every trial in trialResults.

BOOTSTRAPMEAN
Resamples a vector containing sample data computing the mean a given number of times to produce a distribution of the means.

BOOTSTRAPMEDIAN
Resamples a vector containing sample data computing the median a given number of times to produce a distribution of the medians.

BOOTSTRAPSTDDEV
Resamples a vector containing sample data computing the stdev a given number of times to produce a distribution of the stdevs.

DOUBLE_BOOTSTRAP_MEAN_CI
Computes the adjusted and unadjusted lower and upper bounds of a confidence interval using nested bootstraps.

BOXPLOT
Outputs a boxplot of one or more input vectors to the output window.

BREAK
Immediately exits the innermost enclosing REPEAT, FOREACH, WHILE, or UNTIL loop.

BUBBLEGRAPH
Creates a bubble graph of its input vectors in a new graphic window tab.

CHISQUARE
Computes the chi-square statistic from its two input vectors.

CHISQUAREDIST
Returns a given number of random values from the chi-square distribution with given degreesOfFreedom.

CHISQUARE_TABLE
Computes the probabilities and predicted frequencies for the cells of the Chi-square contingency table.

CHISQUARE_TRIALS
Generates the specified number of trials from a Chi-square distribution.

CHOOSETAGSTEST
Returns indexes of elements of inVector that satisfy a specified test. Used inside other subroutines to allow tests for the TAGS command to be passed into other subroutines.

CHOOSETEST
Returns result of a specified test on its arguments. Used to allow tests to be passed into other subroutines.

CLEAN
Removes “missing data” (NaN) from one or more vectors. If CLEAN has more than one argument, if any argument contains an NaN at some position, CLEAN removes elements from all vectors at that same position. This keeps all the remaining elements aligned.

CLEAR
Removes all the elements from one or more vectors.

CLEAROUTPUT
Clears the contents of the Output Window.

CLOSETABS
Closes all graph tabs.

CLUSTERSAMPLE
Generates a cluster sample and stores it into given colNsample vectors.

CLUSTERSAMPLE2

A more computationally efficient version of CLUSTERSAMPLE.

COMBINATIONS
Computes the number of combinations of n items taken k at a time .

COMPLEX
Creates a vector of complex numbers from separate vectors containing the real and imaginary components.

COMPLEX_ADD
Adds corresponding elements of two or more vectors containing complex numbers in rectangular form.

COMPLEX_CONJUGATE
Takes the conjugate of all elements in z. z must be in rectangular form.

COMPLEX_COS
Computes the complex cosine of a complex number in rectangular form.

COMPLEX_COSH
Computes the complex cosh of a complex number in rectangular form.

COMPLEX_COT
Computes the complex cotangent of a complex number in rectangular form.

COMPLEX_COTH
Computes the complex hyperbolic cotangent of a complex number in rectangular form.

COMPLEX_CSC
Computes the complex cosecant of a complex number in rectangular form.

COMPLEX_CSCH
Computes the complex csch of a complex number in rectangular form.

COMPLEX_DIVIDE
Divides corresponding elements of two or more vectors containing complex numbers in rectangular form.

COMPLEX_EXP
Computes the complex exp of a complex number in rectangular form.

COMPLEX_GRAPH
Displays a scatter graph of the complex numbers in the input vector. Complex numbers must be in rectangular form.

COMPLEX_GRAPH_RADIAL
Displays a radial graph of the complex numbers in the input vector. Complex numbers must be in rectangular form.

COMPLEX_IM
Copies the imaginary components of the complex vector z into its result vector, im.

COMPLEX_MAGNITUDE
Computes the magnitudes of each element of a vector containing complex numbers in rectangular form.

COMPLEX_MULTIPLY
Multiplies corresponding elements of two or more vectors containing complex numbers in rectangular form.

COMPLEX_PRINT
Prints multiple vectors containing complex numbers in rectangular form.

COMPLEX_PRINT_POLAR
Prints multiple vectors containing complex numbers in polar form.

COMPLEX_PUT
Copies complex numbers from input vector to the result vector into the positions specified in the pos vector.

COMPLEX_RE
Copies the real components of the complex vector z into its result vector, re.

COMPLEX_SEC
Computes the complex secant of a complex number in rectangular form.

COMPLEX_SECH
Computes the complex hyperbolic cosecant of a complex number in rectangular form.

COMPLEX_SIN
Computes the complex sine of a complex number in rectangular form.

COMPLEX_SINH
Computes the complex sinh of a complex number in rectangular form.

COMPLEX_SPLIT
Splits complex number vector into real and imaginary parts.

COMPLEX_SUBTRACT
Subtracts corresponding elements of two or more vectors containing complex numbers in rectangular form.

COMPLEX_TAKE
Copies complex numbers at specified positions from its input to its result vector. Similar to the TAKE command, but operates on vectors of complex numbers.

COMPLEX_TAN
Computes the complex tangent of a complex number in rectangular form.

COMPLEX_TANH
Computes the complex hyperbolic tangent of a complex number in rectangular form.

COMPLEX_TOPOLAR
Converts a vector containing complex numbers in rectangular form to polar (CIS) form.

COMPLEX_TORECT
Converts a vector containing complex numbers in polar (CIS) form to rectangular form.

CONCAT
Concatenates arguments into a single vector (Same as COPY).

CONST
Creates one or more Named Values. Named Values are Named Constants that have a numerical value.

COPY
Concatenates arguments into a single vector.

CORR
Computes Pearson's product moment correlation coefficient of two vectors.

COS
Computes the cosine of each element of its input vector

COSDEG
Computes the cosine of each element, in degrees, of its input vector.

COSH
Computes the hyperbolic cosine of real x.

COTH
Computes the hyperbolic cotangent of real x.

COUNT
Counts the number of elements that pass a specified test

COVARIANCE
Computes the covariance of two input vectors of equal length

CRAMERS_RULE
Computes the solutions to a system of linear equations.

CSCH
Computes the hyperbolic cosecant of real x.

DATA
Concatenates arguments into a single vector (Same as COPY).

DEBUG
Causes Statistics101 to enter Debug Mode.

DEBUG_AT
Enters Debug mode when counter reaches stopCount

DECLARE
Declares the name and argument list of a subroutine so the subroutine may be invoked in the text of a program prior to its actual definition.

DEDUP
Removes duplicate elements from its input vector.

DELTAS
Returns the differences between adjacent elements.

DERIVATIVE
Computes y1 = dy/dx. Y1 is the derivative.

DETERMINANT
Computes the determinant of matrix up to size 6 by 6.

DIVIDE
Arithmetically divides corresponding elements of its input vectors.

DRAW
Draws howMany items without replacement from the global shuffledUniverse. Used with the RESHUFFLE subroutine.

DRAW2
Draws howMany items from shuffledUniverse without replacement. Used with the RESHUFFLE2 subroutine.

ELSE
Marks the beginning of a set of commands to be executed if an IF command's logical expression evaluates to false and all ELSEIF commands likewise fail.

ELSEIF
Marks the beginning of a set of commands to be executed if its logical expression evaluates to true and the logical expression of its associated IF command and all preceding associated ELSEIF commands evaluate to false.

END
Marks the end of an IF, REPEAT, FOREACH, WHILE, UNTIL, or NEWCMD command block.

ENUM
Creates one or more enumerator constants. Enumerator constants have no numerical value.

EQUAL_ELEMENTS
Returns 0 if all vec's elements are equal; returns 1 otherwise.

EXEC
Submits a command string to the underlying operating system (e.g., Windows, MacOS, Linux) to be executed in a separate process.

EXIT
Terminates the currently running user program.

EXITQ

Asks user whether to terminate or continue the current program.

EXP
Computes the number e (i.e., 2.71828...) raised to the power of each element of the input vector.

EXPONENTIAL
Randomly selects a specified number of values from a specified exponential distribution.

EXTRACT
Draws howMany items from fromVec without replacement.

FACTORIAL
Computes factorials of all elements of vec.

FDIST
Returns a given number of random values from the F distribution with the given degrees of freedom.

FIBONACCI
Computes the Fibonacci series of the given size, where size > 0.

FOREACH
Executes commands between FOREACH and END assigning each element of a given vector one by one to a specified variable.

FRACTION
Copies the fractional part of each element of its inputVector into its result vector.

FUZZ
Sets a range of validity for value comparisons during tests.

FV
Computes future value. 'rate' is a decimal fraction, not a percent.

FV2
Computes future value of variable rates and variable payments. 'rate' is a decimal fraction, not a percent.
GAMMA
Returns the gamma function, gamma(vec) for each real positive element of vec.

GAMMADIST
Returns a given number of random numbers from the gamma distribution Gamma(1,a) for integer a >= 1.

GENERATE
Randomly selects a specified number of elements from a vector, with replacement. (Same as SAMPLE and RANDOM.)

GENERATE_COMBINATIONS

Generates all combinations of data vector's elements taken 'groupSize' elements at a time. For each permutation it invokes a user-written subroutine, PROCESSCOMBINATION.

GENERATE_PERMUTATIONS
Generates all permutations of data vector's elements taken 'groupSize' elements at a time. For each permutation it invokes a user-written subroutine, PROCESSPERMUTATION.

GETARG
Copies the specified optional argument into its result variable.

GETFILEPATH
Retrieves the path information for the file accessed by the most recent READ or WRITE command.

GLOBAL
Declares that the names in its argument list are to be visible within subroutines.

HISTOGRAM
Creates a histogram of one or more vectors in a new graphic window tab.

HISTOGRAMDATA
Computes a histogram from its input vector and puts the results in the remaining vectors. Does not make a plot or graph.

HISTOGRAMPLOT
Prints a text histogram of one or more vectors to the output window.

IF
Allows execution of commands between IF and END if a specified logical expression evaluates to true.

INCLUDE
This command replaces itself with the contents of the file(s) in its argument list.

INCR
Increments each element of vec by one.

INPUT
Prompts the user for input and accepts user's input.

INTEGER
Converts all elements of its input vector to integer by truncation, floor, ceiling, or rounding, depending on the keyword. Default is truncation.

INTEGRAL
Indefinite integral y1 = Integral(y(x)dx). x and y must be same length.

INTEGRALDEF
area is the definite integral of y(x)dx from a to b.

INTERPOLATE_LINEAR
Performs linear interpolation for an x vector containing one or more x values.

IS_ASCENDING
Returns true if input vector is in ascending order.

IS_EVEN
If argument is even, returns true. Otherwise returns false.

IS_SEQUENCE
Returns true if vec is a sequence with common difference of 1, else returns 0.

IS_SEQUENCE2
Returns true if vec is an arithmetic sequence, else 0. Optionally also returns the common difference between elements of vec.

KURTOSIS
Computes the kurtosis of data in the input vector. The KURTOSIS of a normal variable is 3.

KURTOSIS0
Computes the zero-based kurtosis of data in the input vector. The KURTOSIS0 of a normal variable is zero.

LAGGRAPH
Displays a lag graph of a given vector at a given lag interval. A lag graph plots a series of numbers against itself, shifted by one or more positions.

LET
Uses mathematical formula notation to compute a value and assign it to a variable. Allows use of many Statistics101 math command names as unary functions.

LOG
Computes the natural logarithm of each element of its input vector.

LOG10
Computes the base 10 logarithm of each element of its input vector.

LOGNORMAL
Randomly selects a specified number of numbers from a specified lognormal distribution.

LOOKUP
Given the table (sortedKeys, correspondingValues), returns the desiredValues whose keys match the desiredKeys.

MAKECUMDIST
Creates a vector of cumulative probabilities from a vector of probabilities.

MATRIX_ADD

Adds matrices element by element. All matrices must have the same number of rows and the same number of columns.

MATRIX_GETCOL
Returns the column specified by colNum.

MATRIX_GETCOLS
Copies columns from matrixVec one by one into the column arguments.

MATRIX_GETELEMENT
Returns the element at (row,col) of matrixVec.

MATRIX_GETROW
Returns the row specified by rowNum.

MATRIX_GETROWS
Copies rows from matrixVec one by one into the rowVec arguments.

MATRIX_INVERT
Takes the inverse of the given square matrix up to size 6 by 6.

MATRIX_MAKE_BY_COL
Creates a matrix from at least one column vector.

MATRIX_MAKE_BY_ROW
Creates a matrix from at least one row vector.

MATRIX_MULTIPLY
Computes the matrix product of two matrices.

MATRIX_PRINT
Prints the given matrix to the output window.

MATRIX_SUBMATRIX
Returns a submatrix which is matrixVec with the specified row and column removed.

MATRIX_TRANSPOSE
Computes the transpose of matrixVec.

MATRIX_UNIT
Generates a unit matrix. Diagonal elements are all ones. Others are zero.

MAX
Finds the largest value (most positive) in its input vector.

MAXIMUMS
Copies maximum element of all input vectors into same position in result.

MAXSIZE
Not implemented. Was in the original Resampling Stats because of limitations of its memory model. This command is not needed in Statistics101.

MEAN

Computes the mean of a vector.

MEDIAN
Computes the median of a vector.

MIN
Finds the smallest (most negative) value in its input vector.

MINIMUMS
Copies minimum element of all input vectors into same position in result.

MODE
Finds the most frequently occurring value in its input vector.

MODE_CONTINUOUS
Computes mode of a continuous random variable.

MULTINOMIAL
Generates samples from a multinomial distribution defined by the cumDistrib vector.

MULTINOMIAL2
Generates samples from a multinomial distribution defined by the probabilities in the probabilities vector.

MULTIPLES
Computes the number of “multiples” whose sizes satisfy the specified test.

MULTIPLY
Arithmetically multiplies corresponding elements of its input vectors.

NAME
Creates one or more Named Constants.

NEAREST_INDEX
Returns indexes in inVec of elements matching values in valVec or the largest element less than val if there is no match.

NEWARRAY
Creates an array with the given name and dimensions.

NEWCMD
Declares a new user-defined command (subroutine) that can take a fixed or a variable number of arguments.

NORMAL
Randomly selects a specified number of numbers from a specified normal distribution.

NORMALPROB
Calculates the cumulative normal distribution. Given x or z computes p.

NORMALPROBINV
Calculates the inverse cumulative normal distribution. Given p computes z or x.

NORMALQQGRAPH
Displays a Normal Test Plot (or 'Normal Quantile Plot') of a data vector.

NPV
Computes net present value. 'rate' is a decimal fraction, not a percent.

NUMBERS
Concatenates arguments into a single vector (Same as COPY).

OUTPUT
Writes a string and any number of optional numbers to the Output Window or to a specified file.

PARETO
Randomly selects a specified number of numbers from the specified Pareto distribution.

PAUSE
Stops program execution until the user clicks on the Continue button.

PERCENTILE
Computes specified percentiles from an input vector.

PERMUTATIONS
Computes the number of permutations of n items taken k at a time.

POISSON
Randomly selects a specified number of numbers from a specified Poisson distribution.

POWER
Raises each element in the first input vector to the power of the corresponding element in the second input vector.

PREDICT
Estimate a based on the independent variables and coefficients that have been produced by the REGRESS command.

PRINT
Prints the name and contents of one or more vectors to the output window, one vector to a line, or as a table, one vector to a column.

PRODUCT
Computes the product of all the elements of its input vector.

PROGINFO
Prints program variables, constants, and status information to the output window.

PUT
Inserts values from its input vector into its result vector at locations specified by its positions vector.

RANDOM

Randomly selects a specified number of elements from a vector, with replacement. (Same as GENERATE and SAMPLE.)

RANGE
Copies the minimum and maximum values from vec into min and max.

RANKS
Creates a list of the ranks of the elements of its input vector.

READ
Reads a file into one or more result variables (vectors).

RECODE
Replaces with a specified number, any element of the input vector that satisfies a specified test.

REGRESS
Computes the coefficients of the linear regression equation determined by its dependent vector and its independent vector(s).

REMAINDER
Divides the corresponding elements of the two input vectors and puts the remainder in the result vector.

REMOVE
Copies all but the specified elements of its input dataVector into its result vector. See also the TAKE command.

REPEAT
Executes commands between REPEAT and END a specified number of times.

RESHUFFLE
Reshuffles the global, shuffledUniverse, and resets the indexes. Used with the DRAW subroutine.

RESHUFFLE2
Reshuffles the given universe and resets the indexes. Used with the DRAW2 subroutine.

REVERSE
Reverses the order of inVec's elements.

ROTATE
Rotates the elements of the input vector right or left by the specified number of places.

ROUND
Rounds each element of its input vector to the specified number of decimal places.

ROUNDALL
Applies the ROUND command to all argument vectors, in place.

RUNS

Computes the number of runs (consecutive equal numbers) whose lengths satisfy the specified test.

SAMPLE
Randomly selects a specified number of elements from a vector, with replacement. (Same as GENERATE and RANDOM.)

SCALARIZE
Computes a single number by concatenating all the elements of its input vector.

SCATTERGRAPH
Displays a linear or log scatter graph of its input vectors in a graphical tab in the Stastistics101 Output window

SCORE
Accumulates the results of random trials in a scoring vector.

SCORECOORD
Applies SCORE command to multiple vector pairs.

SEARCH_BINARY
Returns indexes in inVec of elements matching values in valVec. Returns NaN for any element that does not match.

SEARCH_BINARY_DESC
Returns indexes in inVec of elements matching values in valVec. Returns NaN for any element that does not match.

SECH
Computes the hyperbolic secant of real x.

SEED
Sets the seed used by the random number generator and/or selects the algorithm that generates the pseudo-random numbers.

SELECTCOORD
For all input vectors, copies elements at positions satisfying a test on the first input vector into corresponding output vectors.

SELECTCOORD_BY_KEY
Chooses data items from a table's column vectors, corresponding to specified key values.

SET
Creates a vector with a specified number of elements all of the same value as the input number (Can be replaced by COPY N#val).

SHIFT

Shifts the elements of the input vector right or left by the specified number of places. Shifts in fillNumber or zeros to the positions freed by the shift. The size of the result vector will be the same as that of the input vector.

SHUFFLE
Randomly reorders the elements of a vector.

SHUFFLECOORD
Coordinated shuffle, in place, of two or more vectors.

SIGN
Substitutes “-1” for negative elements, “+1” for positive elements into the result vector. By default, zero is considered positive. If keyword signum is present, zero is interpreted as zero.

SIN
Computes the sine of each element of its input vector.

SINDEG
Computes the sine of each element, in degrees, of its input vector.

SINH
Computes the hyperbolic sine of real x.

SIZE
Counts the number of elements contained by the input vector.

SKEWNESS
Computes the skewness of data in the input vector.

SMA
Computes simple moving average of inVec over n elements.

SORT
Sorts the elements of the input vector in ascending or descending order.

SORTCOORD
Coordinated sort, in place, of two or more vectors

SORTCOORD_DESC
Descending coordinated sort, in place, of two or more vectors.

SQRT
Computes the square root of each element of the input vector.

SQUARE
Computes the square of each element of the input vector.

STRATASAMPLE
Performs stratified sample on given stratum vectors.

STDEV
Computes the standard deviation of a vector.

STRING
Concatenates string literals, string variables and/or vector variables into one string variable.

STRING_COMPARE
Compares two strings, returning zero if they are equal, a negative number if the first is less than the second, a positive number if the first is greater than the second.

STRING_REPLACE
Returns a new string resulting from replacing all occurrences (or the first) of a regular expression match in the input string with a given replacement string.

SUBTRACT
Arithmetically subtracts corresponding elements of its input vectors.

SUM
Computes the sum of all the elements of its input vector.

SUMABSDEV
Computes the sum of the absolute differences between its two input vectors.

SUMSQRDEV
Computes the sum of the squared deviations of its first input vector's elements versus its second vector's elements.

TAGS
Computes a list of the positions of the elements of the input vector that pass a test.

TAGSORT
Computes a vector whose element values, in order, are the positions of the elements of its input vector as if the input vector were sorted in ascending or descending order.

TAKE
Copies specified elements from its input vector into its result vector. See also the REMOVE command.

TAKECOORD
For any number of vector pairs, copies data by position set by tags from one vector of a pair to the second of the pair.

TAN
Computes the tangent of each element of its input vector.

TANDEG
Computes the tangent of each element, in degrees, of its input vector.

TANH
Computes the hyperbolic tangent of real x.

TDIST

Returns a given number of random values from Student's t distribution with given degreesOfFreedom.

TIME
Reads the system clock and puts the time, in milliseconds, into its result vector.

TIMEPLOT
Prints a timeplot of its input vector on the Statistics101 Output Window.

TOPOLAR
Converts a point described in rectangular coordinates to polar coordinates.

TODEG
Converts each element of its input vector from radians to degrees.

TORAD
Converts each element of its input vector from degrees to radians.

TORECT
Converts a point described in polar coordinates to rectangular coordinates.

TRIANGLE
Generates a given number of random numbers drawn from the Triangle distribution whose minimum x value is minX, most likely value is modeX, and maximum value is maxX.

TRIANGLEPROBDENSITY
Computes Y, the probability density at X, for a triangle distribution with minimum value minX, most likely value modeX, and maximum value maxX.

TRIMMED_MEAN
Computes the trimmed mean of the input vector, given the percentage to be trimmed.

UNIFORM
Selects a specified number of values randomly from the uniform distribution with the specified lower and upper limits.

UNTIL
Executes commands between UNTIL and END until the specified logical expression evaluates to true.

URN
Concatenates arguments into a single vector (Same as COPY).

VARIANCE
Computes the variance of a vector.

WEED
Discards those values from its input vector that satisfy the specified test.

WEIBULL

Randomly selects a specified number of numbers from the specified Weibull distribution.
WHILE
Executes commands between WHILE and END as long as the specified logical expression evaluates to true.

WRITE
Writes its input vector(s) into a file or to the Statistics101 output window according to optional format specifications.

XYGRAPH
Displays an X-Y linear or log graph of its input vectors in a graphical tab in the Stastistics101 Output window

XYPLOT
Prints an X-Y linear or log plot of its input vectors to the Statistics101 Output Window.

Appendix 3: Commands and Subroutines Listed by Category

In this table, the commands are shown in normal type and the subroutines are shown in *Italic* type.

The command and subroutine names that are shaded may be used inside a LET command. Some of the shaded commands, such as INTEGER and SORT, have options that are chosen using a keyword. Since keywords are not allowed in the LET command, those options are given their own special names to be used within LET as if they were ordinary commands. Those names are INTEGER_CEILING, INTEGER_FLOOR, INTEGER_ROUND, INTEGER_TRUNCATE, RANKS_DESCENDING, SIGN_SIGNUM, SORT_DESCENDING, STDEV_POP, TAGSORT_DESCENDING, VARIANCE_POP. They are not listed in the following table because they are not full-fledged commands and are only valid within the LET command. See the entry for LET in the *Statistics101* help document (menu item Help>Help) for full information on each name.

The easiest way to find a command or subroutine when you don't know its exact name or if it even exists is to do a keyword search using the Command/Subroutine Index, which is available from the *Statistics101* program's Help menu, Help>Command/Subroutine Index or by pressing the F3 function key..

Bootstrap

BOOTSTRAPMEAN
BOOTSTRAPMEDIAN
BOOTSTRAPSTDEV
DOUBLE_BOOTSTRAP_MEAN_CI

Calculus

DERIVATIVE
INTEGRAL
INTEGRALDEF

Complex Math

COMPLEX
COMPLEX_ADD
COMPLEX_CONJUGATE
COMPLEX_COS
COMPLEX_COSH
COMPLEX_COT
COMPLEX_COTH
COMPLEX_CSC
COMPLEX_CSCH
COMPLEX_DIVIDE
COMPLEX_EXP
COMPLEX_GRAPH
COMPLEX_GRAPH_RADIAL
COMPLEX_IM
COMPLEX_MAGNITUDE
COMPLEX_MULTIPLY
COMPLEX_PRINT
COMPLEX_PRINT_POLAR
COMPLEX_PUT
COMPLEX_RE
COMPLEX_SEC
COMPLEX_SECH
COMPLEX_SIN
COMPLEX_SINH
COMPLEX_SPLIT
COMPLEX_SUBTRACT
COMPLEX_TAKE
COMPLEX_TAN
COMPLEX_TANH
COMPLEX_TOPOLAR
COMPLEX_TORECT
TOPOLAR
TORECT

Control Flow

BREAK
DEBUG
DEBUG_AT
ELSE
ELSEIF
END
EXIT
EXITQ
FOREACH
IF
PAUSE
REPEAT

UNTIL
WHILE

Coordinated Operations

LOOKUP
SCORECOORD
SELECTCOORD
SELECTCOORD_BY_KEY
SHUFFLECOORD
SORTCOORD
SORTCOORD_DESC
TAKECOORD

Data Entry[1]

CONCAT
CONST
COPY
DATA
ENUM
INPUT
NAME
NEWARRAY
NUMBERS
READ
SET
URN

Distributions

BETA
BETA_01
BINOMIALSAMPLE
BINOMIALTRIALS
CHISQUAREDIST
EXPONENTIAL
FDIST
GAMMADIST
LOGNORMAL
MAKECUMDIST
MULTINOMIAL
MULTINOMIAL2
NORMAL
PARETO
POISSON
TDIST
TRIANGLE
TRIANGLEPROBDENSITY
UNIFORM
WEIBULL

Financial

FV
FV2
NPV
SMA

Input/Output

GETFILEPATH
INPUT
OUTPUT
PRINT
READ
WRITE

Math Functions

ABS
ACOS
ACOSDEG
ADD
ASIN
ASINDEG
ATAN
ATANDEG
COS
COSDEG
COSH
COTH
CSCH
DIVIDE
EXP
FACTORIAL
FRACTION
GAMMA
INCR
INTEGER
INTERPOLATE_LINEAR
IS_EVEN
LET
LOG
LOG10
MAX
MAXIMUMS
MIN
MINIMUMS

[1]CONCAT, COPY, DATA, NUMBERS, and URN are really different names for the same command. You can choose among these names depending on context to make your program more clear. The SET command can be replaced by a COPY command and "multiple literals".

MULTIPLY
POWER
PRODUCT
RANGE
REMAINDER
ROUND
ROUNDALL
SECH
SIGN
SIN
SINDEG
SINH
SMA
SQRT
SQUARE
SUBTRACT
SUM
SUMABSDEV
SUMSQRDEV
TAN
TANDEG
TANH
TODEG
TORAD

Matrix Math

CRAMERS_RULE
DETERMINANT
MATRIX_ADD
MATRIX_GETCOL
MATRIX_GETCOLS
MATRIX_GETELEMENT
MATRIX_GETROW
MATRIX_GETROWS
MATRIX_MAKE_BY_COL
MATRIX_MAKE_BY_ROW
MATRIX_MULTIPLY
MATRIX_PRINT
MATRIX_SUBMATRIX
MATRIX_TRANSPOSE
MATRIX_UNIT
MATRIX_INVERT

Plotting/Graphing

AUTOCORRGRAPH
BOXPLOT
BUBBLEGRAPH
COMPLEX_GRAPH
COMPLEX_GRAPH_RADIAL
HISTOGRAM
HISTOGRAMDATA
HISTOGRAMPLOT
LAGGRAPH
NORMALQQGRAPH
SCATTERGRAPH
TIMEPLOT
XYGRAPH
XYPLOT

Sampling/ Randomizing[2]

CLUSTERSAMPLE
DRAW
DRAW2
EXTRACT
GENERATE
RANDOM
REMOVE
RESHUFFLE
RESHUFFLE2
SAMPLE
SHUFFLE
STRATASAMPLE
TAKE

Search

LOOKUP
NEAREST_INDEX
SEARCH_BINARY
SEARCH_BINARY_DESC

Series

FIBONACCI
IS_ASCENDING
IS_SEQUENCE

Sort

SORT
SORTCOORD
SORTCOORD_DESC
TAGSORT

Statistics Functions

AUTOCORR
BINOMIALPROB
CHISQUARE

[2] GENERATE, RANDOM, and SAMPLE are really different names for the same command. You can choose among these names depending on context to make your program more clear.